全国优秀畅销图书奖

全国普通高等学校优秀教材一等奖

普通高等教育"十一五"国家级规划教材

清華大學 计算机系列教材

张尧学 宋虹 张高 编著

# 计算机操作系统教程

## （第4版）



清华大学出版社

清华大学计算机系列教材

# 计算机操作系统教程

## （第 4 版）

张尧学　宋　虹　张　高　编著

清 华 大 学 出 版 社

北　京

## 内 容 简 介

操作系统是现代计算机系统中必不可少的基本系统软件，也是计算机专业的必修课程和从事计算机应用人员必不可少的知识。

本书是编著者在清华大学计算机系多年教学和科研的基础上对第3版改编而成的，全书共12章，主要内容包括操作系统用户界面、进程与线程管理、处理机管理、内存管理、文件系统与设备管理等基本原理及 Linux 和 Windows 两个主流操作系统的内核介绍。

与第3版相比，本书进一步深入浅出地对操作系统的基本原理进行了描述，并且，本书更进一步强调了学生对当前主流操作系统的应有了解。因此，本书在第3版 Linux 2.4 和 Windows NT 操作系统实例的基础上，补充了有关 Linux 2.6、Windows NT 6.0 以及嵌入式操作系统的相关知识。

本书可作为高等院校计算机专业或相关专业操作系统课程的教材，也可供有关科技人员自学或参考。



**图书在版编目(CIP)数据**

计算机操作系统教程 / 张尧学，宋虹，张高编著．—4版．—北京：清华大学出版社，2013
清华大学计算机系列教材
ISBN 978-7-302-33668-6

Ⅰ．①计…　Ⅱ．①张…　②宋…　③张…　Ⅲ．①操作系统—高等学校—教材　Ⅳ．①TP316

中国版本图书馆 CIP 数据核字(2013)第 206342 号

**责任编辑**：白立军　战晓雷
**封面设计**：常雪影
**责任校对**：梁　毅
**责任印制**：何　芊

**出版发行**：清华大学出版社
**网　址**：http://www.tup.com.cn，http://www.wqbook.com
**地　址**：北京清华大学学研大厦 A 座　　**邮　编**：100084
**社 总 机**：010-62770175　　**邮　购**：010-62786544
**投稿与读者服务**：010-62776969，c-service@tup.tsinghua.edu.cn
**质量反馈**：010-62772015，zhiliang@tup.tsinghua.edu.cn
**课件下载**：http://www.tup.com.cn，010-62795954
**印 刷 者**：北京富博印刷有限公司
**装 订 者**：北京市密云县京文制本装订厂
**经　销**：全国新华书店
**开　本**：185mm×260mm　　**印　张**：19.75　　**字　数**：490千字
**版　次**：1993年10月第1版　2013年10月第4版　　**印　次**：2013年10月第1次印刷
**印　数**：1～3000
**定　价**：34.50元

产品编号：040257-01

# 第 4 版前言

计算机技术的飞速发展超过了人们的想象。操作系统作为一门计算机的基础课程，无论是对计算机等信息技术专业的学生或研究人员，还是对一般计算机应用人员而言，都是非常有益和重要的。

本书自 1993 年出版以来，得到了广大读者的支持和厚爱。特别是 1999 年第 2 版之后，本书更得到了许多学校的老师和学生们的支持。这令编著者非常感动。在前 3 版的基础上，结合嵌入式系统技术及操作系统技术的发展，我们对本书内容进行了新的补充。

本书的改编考虑了如下几个事实：

首先，这是一本讲授操作系统基本原理的本科生教材，因此讲授内容不宜过深过细，而重在强调“为什么”、“是什么”和“怎样做”。因此，本书未在基本概念及基本原理方面进行变动。

其次，考虑到 Linux 2.6 和 Windows NT 6.0 内核版本是目前正在使用的主流操作系统内核版本，本书的操作系统实例在第 3 版中的 Linux 2.4 和 Windows NT 的基础上，适当补充了 Linux 2.6 和 Windows NT 6.0 内核版本的一些内容。

第三，考虑到嵌入式系统的发展，本书在最后增加了嵌入式操作系统的管理机制、嵌入式操作系统的集成开发环境及开发过程的内容，并对实验和习题进行了改写。

本书共 12 章。第 1 章简要介绍操作系统的基本概念、功能、分类以及发展历史等。第 2 章主要讨论操作系统的两种界面和简单的使用操作方法。第 3 章介绍进程与线程管理的有关概念和技术。第 4 章主要介绍处理机管理和调度策略。调度策略与算法主要用于处理机管理，但在交换区等其他资源分配时也被大量使用。第 5 章介绍存储管理，包括分区、分页、分段和段页式管理等。作为进程管理与存储管理的实例，第 6 章和第 7 章分别介绍 Linux 和 Windows NT 的进程与存储管理系统。第 8 章介绍文件系统。第 9 章讲述设备管理技术。第 10 章和第 11 章则在第 8 章与第 9 章的基础上介绍 Linux 和 Windows NT 的文件和设备管理方法。第 12 章简述了嵌入式操作系统的基本原理、嵌入式操作系统的集成开发环境及开发过程。

本书的讲授学时可安排为约 68～76 学时：第 1 章为 2 学时，第 2 章为 4 学时，第 3 章为 8～10 学时，第 4 章为 6 学时，第 5 章为 6～8 学时，第 6 章为 8 学时，第 7 章为 6 学时，第 8 章为 8 学时，第 9 章为 6 学时，第 10 章与第 11 章分别为 4～6 学时和 4 学时，第 12 章为 6～8 学时。教师也可根据自己的教学计划安排学时。

本书第 1 章的 1.1 节至 1.4 节以及 1.7 节由史美林教授编写，第 7 章和第 11 章由微软亚洲研究院张高博士编写，第 12 章由中南大学宋虹编写，第 6 章和第 10 章由红旗 Linux 公司门小燕女士提供了资料，其他章节由张尧学编写。

在本书的改编过程中，清华大学史美林教授和华北水利水电学院朱贵良教授提供了宝贵的意见和修改建议；清华大学杨华杰同志帮助整理和试做了所有习题和实验；还有教育部

领导和同事们对编著者“不务正业”的容忍和给予时间上的便利。多少个节假日不能休息，不能和家庭团聚，但家人们仍然毫无怨言，以最大的爱支持我们的工作，编著者们对他们致以万分的感谢！没有大家的支持，本书的改编是不可能完成的。

由于编著者水平有限，书中难免有错误和不妥之处，恳请广大读者批评指正。

编著者

2013 年 6 月

# 目　　录

# 第1章　绪　　论

计算机发展到今天，从个人计算机到巨型计算机系统，毫无例外都配置一种或多种操作系统。什么是操作系统，它具有什么样的功能等，将在这一章作简要阐述。为了阐明这些问题，扼要地回顾一下操作系统的形成和发展过程是必要的。为便于今后的学习，本章随后介绍操作系统的类型及其特点，以及研究操作系统的几种观点。

## 1.1　操作系统概念

### 什么是操作系统

迄今，任何一个计算机系统都配置一种或多种操作系统。

计算机系统由两部分组成：硬件和软件。计算机硬件通常由中央处理机（运算器和控制器）、存储器、输入设备和输出设备等部件组成，它构成了系统本身和用户作业赖以活动的物质基础和工作环境。

计算机软件包括系统软件和应用软件。系统软件包括操作系统、多种语言处理程序（汇编和编译程序等）、连接装配程序、系统实用程序和多种工具软件等；应用软件是为应用编制的程序。

没有任何软件支持的计算机称为裸机（bare machine），它仅仅构成了计算机系统的物质基础，而实际呈现在用户面前的计算机系统是经过若干层软件改造的计算机。图1.1展示了这种情形。

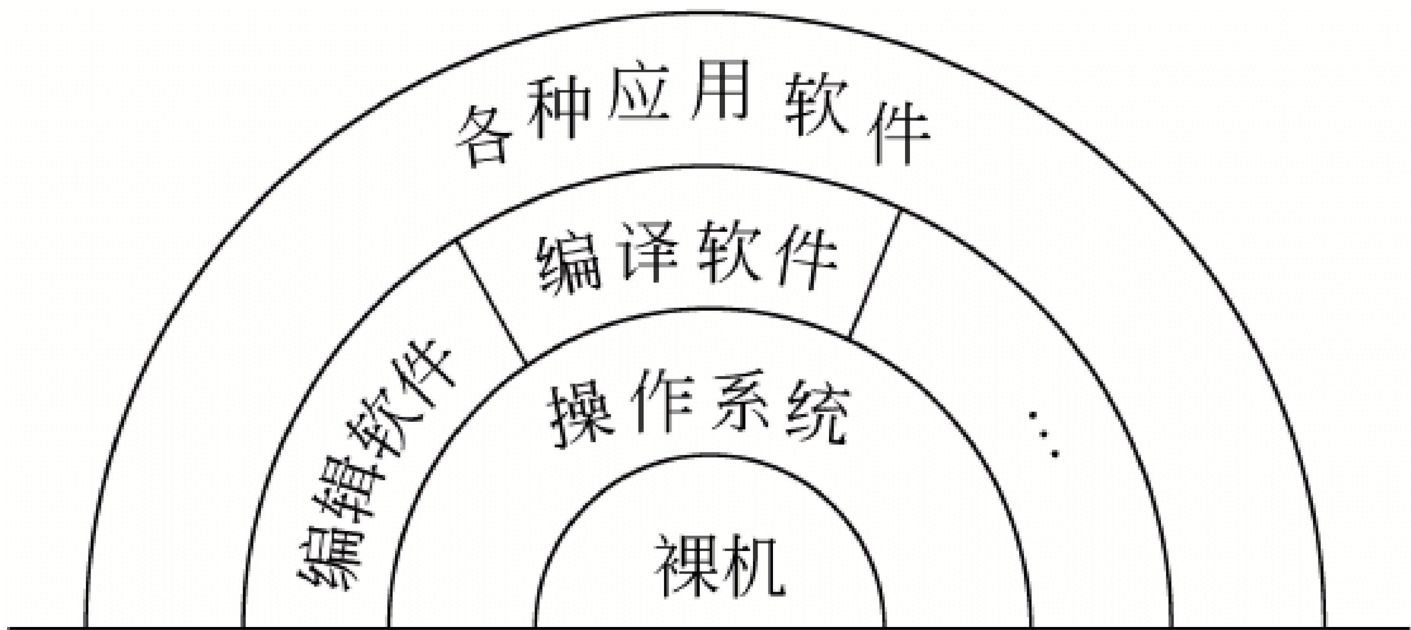


图1.1　操作系统与硬件软件的关系

由图1.1可以看出，计算机的硬件和软件以及应用之间是一种层次结构的关系。裸机在最里层，它的外面是操作系统，操作系统提供的资源管理功能和方便用户的各种服务功能把裸机改造成功能更强、使用更为方便的机器，通常称为虚拟机（virtual machine）或扩展机（extended machine），而各种实用程序和应用程序运行在操作系统之上，它们以操作系统作为支撑环境，同时又向用户提供完成其作业所需的各种服务。

引入操作系统的目的可从三方面来考察。

（1）从用户的观点来看，计算机是为用户提供服务的，计算机所完成的任何工作都是为了满足用户的计算或处理需求。因此，引入操作系统是让计算机为用户提供最好的服务，构建一个用户和计算机之间的和谐交互环境。这要求计算机有一个良好的用户界面，使用户无须了解许多有关硬件和系统软件的细节，能够方便灵活地使用计算机。同时，计算机还能为用户提供一个可靠和安全的服务管理，以保证用户得到可靠安全的服务。

（2）从系统管理人员的观点来看，引入操作系统是为了合理地组织计算机工作流程，管理和分配计算机系统硬件及软件资源，使之能为多个用户高效率地共享。因此，操作系统是

计算机资源的管理者。

(3) 从发展的观点看,引入操作系统是为了给计算机系统的功能扩展提供支撑平台,使之在追加新的服务和功能时更加容易并且不影响原有的服务与功能。

综上所述,可以非形式地把操作系统定义为:操作系统是计算机系统中的一个系统软件,它是这样一些程序模块的集合——它们管理和控制计算机系统中的硬件及软件资源,合理地组织计算机工作流程,以便有效地利用这些资源为用户提供一个具有足够的功能、使用方便、可扩展、安全和可管理的工作环境,从而在计算机与其用户之间起到接口的作用。

操作系统的几个主要特点是:它是一个管理计算机软硬件资源的系统软件,它为用户提供尽可能多的服务,它的管理过程根据用户要求不同而有所不同,但主要是为了让用户高效率地共享计算机软硬件资源,但又要保证其可靠性、安全性、可用性和可管理性。

## 1.2 操作系统的历史

为了更好地理解操作系统的基本概念、功能和特点,本节首先回顾操作系统形成和发展的历史过程。

操作系统是由于客观的需要而产生的,它伴随着计算机技术本身及其应用的日益发展而逐渐发展和不断完善。它的功能由弱到强,在计算机系统中的地位不断提高。至今,它已成为计算机系统中的核心,无一计算机系统是不配置操作系统的。

由于操作系统历来跟运行其上的计算机组成与体系结构休戚与共,因此下面考察各代计算机,看看它们的操作系统是什么样子,具有哪些功能和特征。

人们通常按照器件工艺的演变把计算机发展过程分为4个阶段。

1946年至20世纪50年代末:第一代,电子管时代,无操作系统。

20世纪50年代末至60年代中期:第二代,晶体管时代,批处理系统。

20世纪60年代中期至70年代中期:第三代,集成电路时代,多道程序设计。

20世纪70年代中期至20世纪末:第四代,大规模和超大规模集成电路时代,分时系统。

21世纪初开始,以移动、分布和网络计算为代表,现代计算机正向着普适计算、网格计算以及巨型、微型、并行、分布、网络化、智能化和生物信息化几个方面发展着。

适应上述计算机发展过程,操作系统经历了如下的发展过程:手工操作阶段(无操作系统)、批处理、执行系统、多道程序系统、分时系统、实时系统、通用操作系统、网络操作系统和分布式操作系统等。

### 1.2.1 手工操作阶段

在第一代计算机时期,构成计算机的主要元器件是电子管,计算机运算速度慢(只有几千次/秒),没有操作系统,甚至没有任何软件。用户直接用机器语言编制程序,并在上机时独占全部计算机资源。用户既是程序员,又是操作员。上机完全是手工操作:先把程序纸带(或卡片)装上输入机,然后启动输入机把程序和数据送入计算机,接着通过控制台开关启动程序运行。计算完毕,打印机输出计算结果,用户取走并卸下纸带(或卡片)。第二个用户程序上机,照此办理。这种由一道程序独占机器且有人工操作的情况,在计算机速度较慢时

是允许的，因为此时计算机所需时间相对较长，手工操作所占比例还不很大。

20世纪50年代后期，计算机的运行速度有了很大提高，从每秒几千次、几万次发展到每秒几十万次、上百万次。这时，手工操作的慢速度和计算机的高速度之间形成矛盾，手工操作与计算机有效运行时间之比大大地加大，这种矛盾已经到了不能容忍的地步。唯一的解决办法是摆脱人的手工操作，实现作业的自动过渡。这样就出现了批处理。

### 1.2.2 早期批处理

在计算机发展的早期阶段，用户上机时需要自己建立和运行作业，并做结束处理。由于没有任何用于管理的软件，所有的运行管理和具体操作都由用户自己承担。每个作业都由许多作业步组成，任何一步的错误操作都可能导致该作业从头开始。在当时，计算机的价格是极其昂贵的，计算机(CPU)的时间是非常宝贵的，尽可能提高CPU的利用率成为十分迫切的任务。

解决的途径有两个：一个是配备专门的计算机操作员，程序员不再直接操作计算机，减少操作错误；另一个是进行批处理(batch processing)，操作员把用户提交的作业分类，把一批作业编成一个作业执行序列。每一批作业将有专门编制的监督程序(monitor)自动依次处理。

早期的批处理可分为两种方式。

**1. 联机批处理**

慢速的输入输出(I/O)设备和主机直接相连。作业的执行过程如下：

(1) 用户提交作业，包括作业程序、数据以及用作业控制语言编写的作业说明书。

(2) 作业被做成穿孔纸带或卡片。

(3) 操作员有选择地把若干作业合成一批，通过输入设备(纸带输入机或读卡机)把它们存入磁带。

(4) 监督程序读入一个作业(若系统资源能满足该作业要求)。

(5) 从磁带调入汇编程序或编译程序，将用户作业源程序翻译成目标代码。

(6) 连接装配程序把编译后的目标代码及所需的子程序装配成一个可执行程序。

(7) 启动执行。

(8) 执行完毕，由善后处理程序输出计算结果。

(9) 再读入一个作业，重复(5)～(9)步。

(10) 一批作业完成，返回到(3)，处理下一批作业。

这种联机批处理方式解决了作业自动转接的问题，从而减少了作业建立和人工操作时间。但是在作业的输入和执行结果的输出过程中，主机CPU仍处在等待状态，这样慢速的输入输出设备和快速主机之间仍处于串行工作，CPU的时间仍有很大的浪费。

**2. 脱机批处理**

脱机批处理方式的显著特征是增加一台不与主机直接相连而专门用于与输入输出设备打交道的卫星机，如图1.2所示。

卫星机的功能如下：

(1) 输入设备通过卫星机把作业输入到输入带。

(2) 输出带通过卫星机将作业执行结果输出到输出设备。

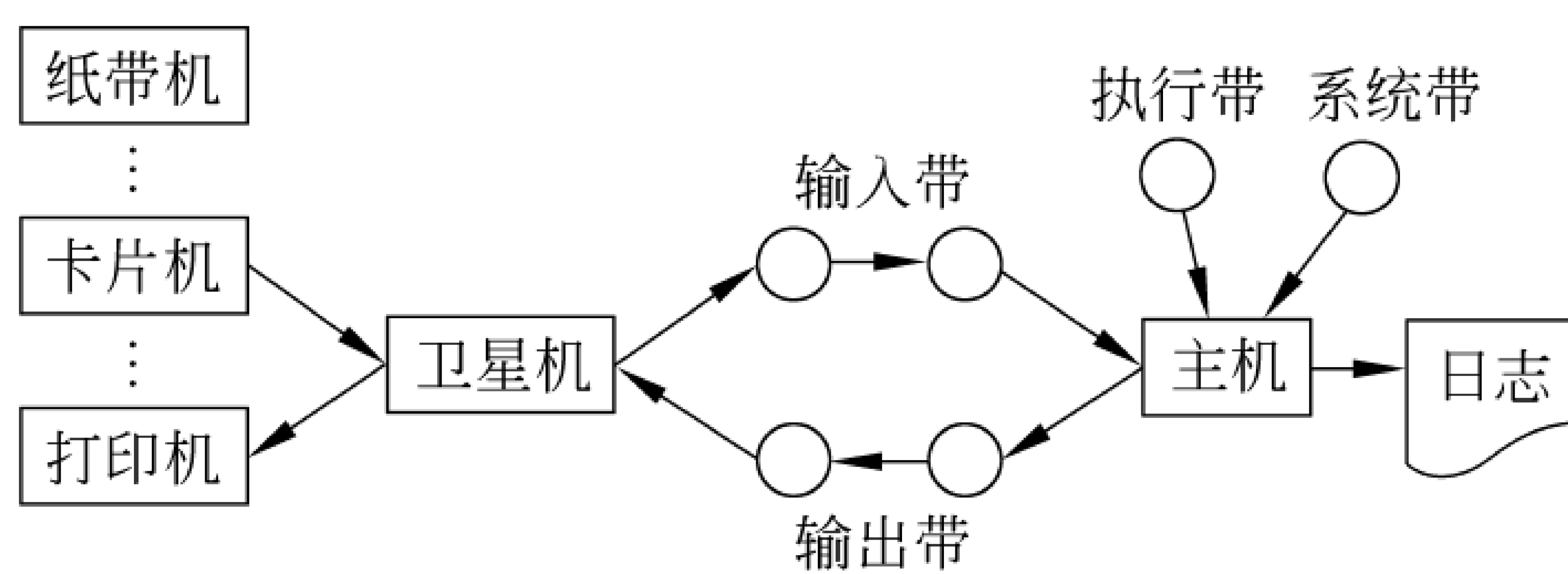



图 1.2　早期脱机批处理模型

这样,主机不是直接与慢速的输入输出设备打交道,而是与速度相对较快的磁带机发生关系。主机与卫星机可以并行工作,二者分工明确,以充分发挥主机的高速计算能力。因此脱机批处理和早期联机批处理相比大大提高了系统的处理能力。

批处理出现于 20 世纪 50 年代末到 60 年代初,它是为了提高主机的使用效率,在解决人机矛盾(主机高速度和输入输出设备的慢速度的矛盾)的过程中逐步发展起来的。它的出现促使了软件的发展。再有重要的是监督程序,它管理作业的运行——负责装入和运行各种系统处理程序,如汇编程序、编译程序、连接装配程序和程序库(如输入输出标准程序等);完成作业的自动过渡,同时也出现了程序覆盖等程序设计技术。

批处理克服了手工操作的缺点,实现了作业的自动过渡,改善了主机 CPU 和输入输出设备的使用情况,提高了计算机系统的处理能力。但它仍有些缺点:磁带需人工拆装,既麻烦又易出错;而另一个更重要的问题是系统的保护。下面先对在监督程序管理下的解题过程进行分析,如图 1.3 所示。

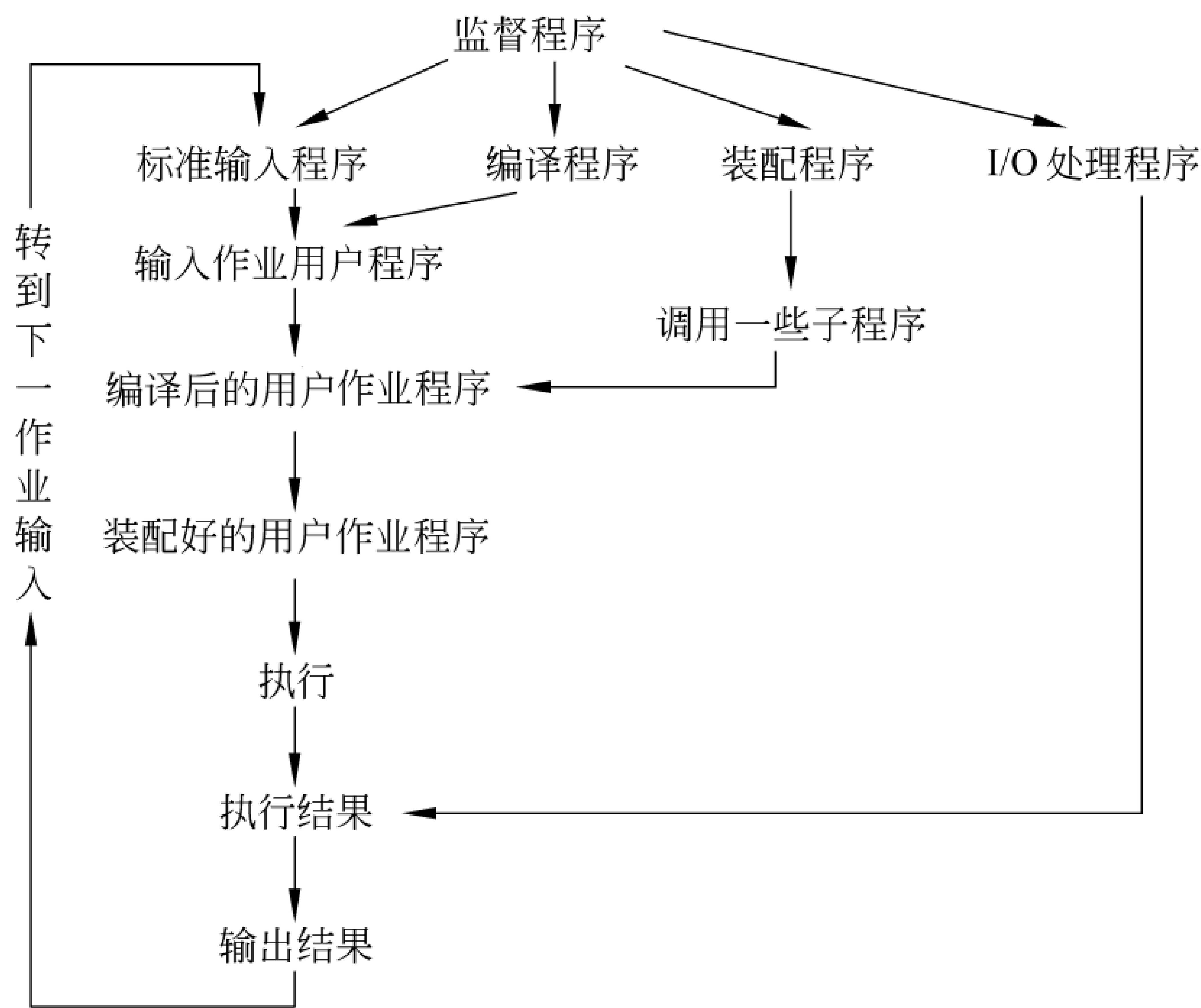



图 1.3　监督程序管理下的解题过程

在进行批处理过程中,监督程序、系统程序和用户程序之间存在着一种调用关系,任何一个环节出问题,整个系统都会停顿;用户程序也可能会破坏监督程序和系统程序,这时,只有操作员进行干预才能恢复。20 世纪 60 年代初期,硬件获得了两方面(即通道和中断技术)的进展,导致操作系统进入执行系统阶段。

通道是一种专用处理部件，它能控制一台或多台输入输出设备工作，负责输入输出设备与主存之间的信息传输。它一旦被启动就能独立于CPU运行，这样可使CPU和通道并行操作，而且CPU和多种输入输出设备也能并行操作。中断是指当主机接到外部信号（如输入输出设备完成信号）时，马上停止原来工作，转去处理这一事件，处理完毕后，主机回到原来的断点继续工作。

借助于通道、中断技术和输入输出可在主机控制下完成批处理。这时，原来的监督程序的功能扩大了，它不仅要负责作业运行的自动调度，而且还要提供输入输出控制功能。这个发展了的监督程序常驻内存，称为执行系统（executive system）。执行系统实现的也是输入输出联机操作，和早期批处理系统不同的是：输入输出工作是由在主机控制下的通道完成的。主机和通道、主机和输入输出设备都可以并行操作。用户程序的输入输出工作都由系统执行而没有人工干预，由系统检查其命令的合法性，以避免不合法的输入输出命令造成对系统的影响，从而提高系统的安全性。此时，除了输入输出中断外，其他中断如算术溢出和非法操作码中断等可以克服错误停机，而时钟中断可以解决用户程序中出现的死循环等。

许多成功的批处理系统在20世纪50年代末至60年代初出现，典型的操作系统是FMS（FORTRAN Monitor System，FORTRAN监督系统）和IBM 7094机上的IBM操作系统IBSYS。执行系统实现了主机、通道和输入输出设备的并行操作，提高了系统效率，方便用户对输入输出设备的使用。但是，这时计算机系统运行的特征是单道顺序地处理作业，即用户作业仍然是一道一道作业顺序处理。因此可能会出现两种情况：对于以计算为主的作业，输入输出量少，外围设备空闲；然而对于以输入输出为主的作业，又会造成主机空闲。这样总的来说，计算机资源使用效率仍然不高。因此操作系统进入了多道程序阶段：多道程序合理搭配，交替运行，充分利用资源，提高效率。

### 1.2.3 多道程序系统

上述批处理系统，每次只调用一个用户作业程序进入内存并运行，称为单道运行。图1.4(a)给出了单道程序工作示例，图1.4(b)给出了多道程序工作示例。

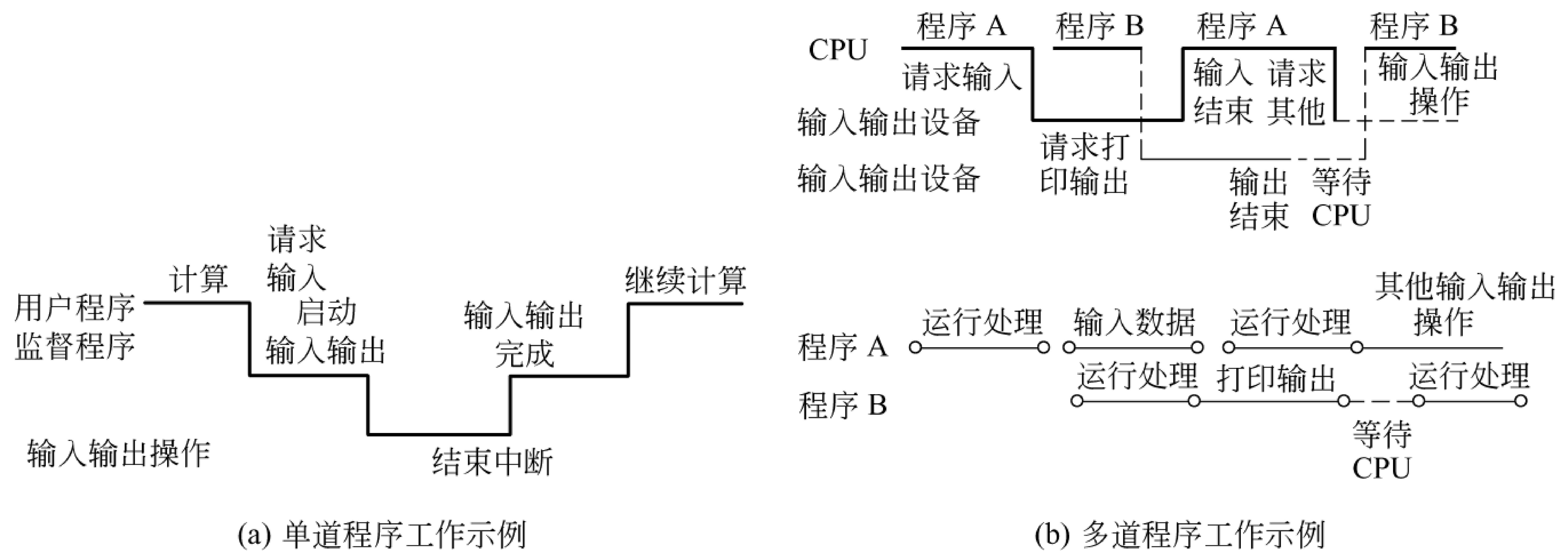



图1.4 程序工作示例

在单处理机系统中，多道程序运行的特点如下：

(1) 多道。计算机内存中同时存放几道相互独立的程序。

(2) 宏观上并行。同时进入系统的几道程序都处于运行过程中,即它们先后开始了各自的运行,但都未运行完毕。

(3) 微观上串行。实际上,各道程序轮流使用 CPU,交替执行。

在批处理系统中采用多道程序设计技术,就形成了多道批处理系统。要处理的许多作业存放在外部存储器中,形成作业队列,等待运行。当需要调入作业时,将由操作系统中的作业调度程序对外存中的一批作业,根据其对资源的要求和一定的调度原则,调几个作业进入内存,让它们交替运行。当某个作业完成后,再调入一个或几个作业。采用这种处理方式,在内存中总是同时存在几道程序,系统资源得到比较充分的利用。

在多道程序系统中,要解决这样一些技术问题:

(1) 并行运行的程序要共享计算机系统的硬件和软件资源,既有对资源的竞争,又必须相互同步。因此同步与互斥机制成为操作系统设计中的重要问题。

(2) 随着多道程序的增加,出现了内存不够用的问题,提高内存的使用效率也成为关键。因此出现了诸如覆盖技术、对换技术和虚拟存储技术等内存管理技术。

(3) 由于多道程序存在于内存,为了保证系统程序存储区和各用户程序存储区的安全可靠,提出了内存保护的要求。

多道程序系统的出现标志着在操作系统渐趋成熟的阶段先后出现了作业调度管理、处理机管理、存储器管理、外部设备管理和文件系统管理等功能。

### 1.2.4 分时操作系统

批处理方式下,用户以脱机操作方式使用计算机,用户在提交作业以后就完全脱离了自己的作业,在作业运行的过程中,不管出现什么情况,用户都不能加以干预,只有等该批作业处理结束,用户才能得到计算结果。根据结果再作下一步处理,若有错,还得重复上述过程。这种操作方式的好处是计算机效率高。不过,用户十分留恋手工操作阶段的联机工作方式,独占计算机,并直接控制程序运行。但独占计算机方式会造成资源效率低。既能保证计算机效率,又能方便用户使用,成为一种新的追求目标。20 世纪 60 年代中期,计算机技术和软件技术的发展使这种追求成为可能。由于 CPU 速度不断提高和采用分时技术,一台计算机可同时连接多个用户终端,而每个用户可在自己的终端上联机使用计算机,好像自己独占计算机一样。

所谓分时技术,就是把处理机的运行时间分成很短的时间片,按时间片轮流把处理机分配给各联机作业使用。若某个作业在分配给它的时间片内不能完成其计算,则该作业暂时中断,把处理机让给另一作业使用,等待下一轮时再继续其运行。由于计算机速度很快,作业运行轮转得很快,给每个用户的印象是好像他独占了一台计算机。而每个用户可以通过自己终端向系统发出各种操作控制命令,完成作业的运行。

多用户分时操作系统是当今计算机操作系统中最普遍使用的一类操作系统。

### 1.2.5 实时操作系统

20 世纪 60 年代中期计算机进入第三代,计算机的性能和可靠性有了很大提高,造价亦大幅度下降,导致计算机应用越来越广泛。计算机应用于工业过程控制、军事实时控制等领域就形成了各种实时系统。实时操作系统是以在允许的时间范围之内做出响应为特征的。

它要求计算机对于外来信息能以足够快的速度进行处理，并在被控对象允许时间范围内做出快速响应，其响应时间要求在秒级、毫秒级甚至微秒级或更小。实时操作系统在嵌入式计算中得到了越来越广泛的应用。特别是移动计算等非 PC、PDA（个人数字助理）和手机等新设备的出现，更加强了这一趋势。

例如，随着移动通信进入 3G 时代，诺基亚等公司研制的 Symbian 手机操作系统、微软公司研制的 Windows Mobile、Google 公司等研制的 Android 系统、近年崛起的操作系统新秀 Linux 等都已有了巨大的市场和用户群体。

### 1.2.6 通用操作系统

多道批处理系统和分时系统的不断改进，实时系统的出现及其应用日益广泛，致使操作系统日益完善。在此基础上，出现了通用操作系统，它可以同时兼有多道批处理、分时、实时处理的功能，或其中两种以上的功能。例如，将实时处理和批处理相结合构成实时批处理系统，在这样的系统中，首先保证优先处理实时任务，插空进行批作业处理。通常把实时任务称为前台作业，批作业称为后台作业。将批处理和分时处理相结合可构成分时批处理系统，在保证分时用户的前提下，没有分时用户时可进行批量作业的处理。同样，分时用户和批处理作业可按前后台方式处理。

从 20 世纪 60 年代中期开始，国际上开始研制大型通用操作系统。这些系统试图达到功能齐全、可适应各种应用范围和操作方式变化多端的环境的目标。但是这些系统本身很庞大，不仅付出了巨大的代价，而且由于系统过于复杂和庞大，在解决其可靠性、可维护性、可理解性和开放性等方面都遇到了很大的困难。相比之下，UNIX 操作系统却是一个例外。这是一个通用的多用户分时交互型的操作系统。它首先建立的是一个精干的核心，而其功能却足以与许多大型的操作系统相媲美，在核心层以外可以支持庞大的软件系统，它很快得到应用和推广并不断完善，对现代操作系统有着重大的影响。目前广泛使用的各种工作站级的操作系统，例如 SUN 公司的 Solaris 和 IBM 公司的 AIX 等都是基于 UNIX 的操作系统。Microsoft 公司的 Windows 系列操作系统，其主要原理也是基于 UNIX 系统的。另外，目前广为流传的 Linux 系统也是从 UNIX 演变而来的。

至此，操作系统的基本概念、功能、基本结构和组成都已形成并渐趋完善。

### 1.2.7 操作系统的进一步发展

进入 20 世纪 80 年代，随着大规模集成电路工艺技术的飞跃发展以及微处理机的出现和发展，掀起了计算机大发展大普及的浪潮。一方面迎来了个人计算机的时代，同时又向计算机网络、分布式处理、巨型计算机和智能化方向发展。操作系统有了进一步的发展：

- 个人计算机上的操作系统，例如 Windows 操作系统系列；
- 嵌入式操作系统，例如 Symbian 操作系统；
- 网络操作系统；
- 分布式操作系统；
- 智能化操作系统。

# 1.3 操作系统的基本类型

通过上一节的讨论已知，随着计算机技术和软件技术长期发展，已形成了各种类型的操作系统，以满足不同的应用要求。根据其使用环境和对作业处理方式的不同，操作系统的基本类型可划分为如下几种：

(1) 批处理操作系统(batch processing operating system)；

(2) 分时操作系统(time sharing operating system)；

(3) 实时操作系统(real time operating system)；

(4) 个人计算机操作系统(personal computer operating system)；

(5) 网络操作系统(network operating system)；

(6) 分布式操作系统(distributed operating system)。

下面对它们作概要的说明。

## 1.3.1 批处理操作系统

批处理操作系统是一种早期的大型机用操作系统。不过，现代操作系统大都具有批处理功能。图 1.5 给出了批处理系统中的作业处理步骤及状态。

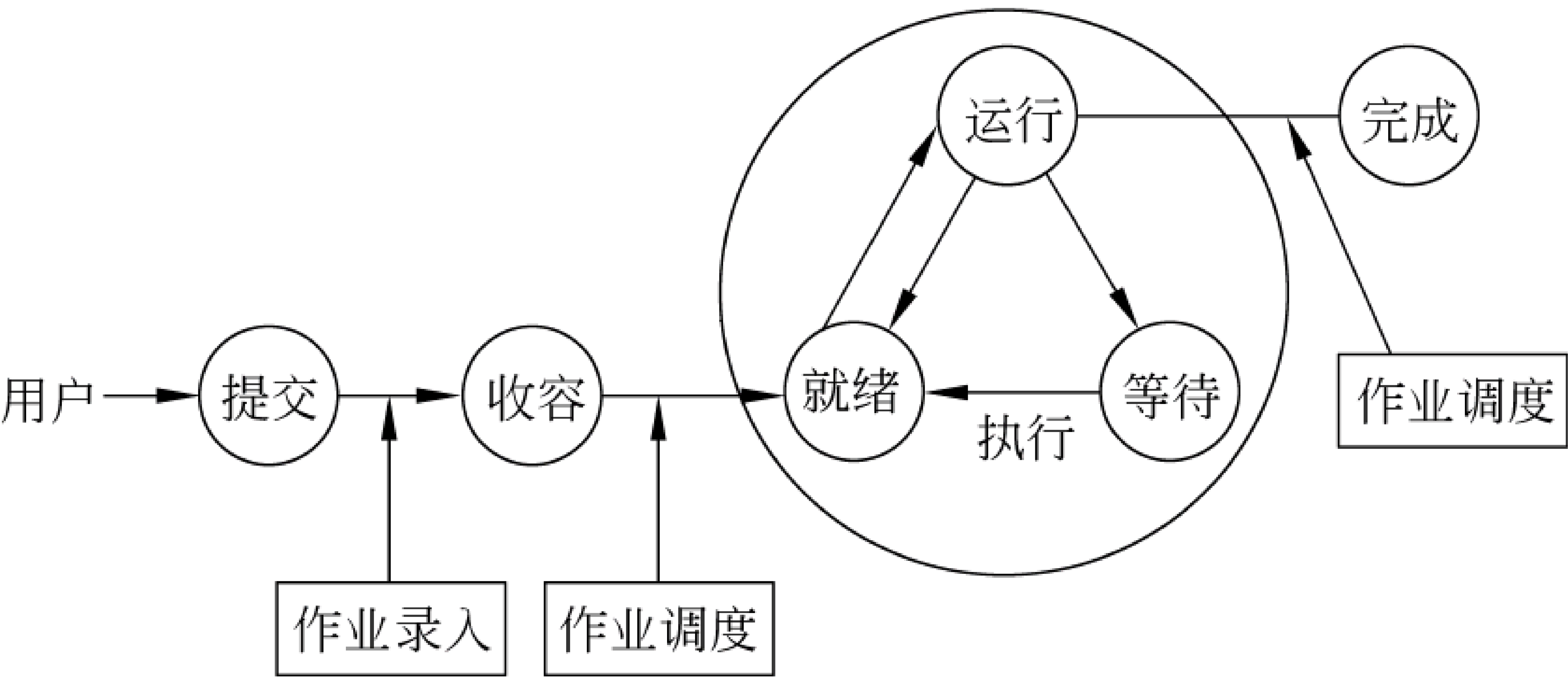



图 1.5 批处理系统中作业处理步骤及状态

批处理系统的主要特征如下：

(1) 用户脱机使用计算机。用户提交作业之后直到获得结果之前就不再和计算机打交道。作业提交的方式可以是直接交给计算中心的管理操作员，也可以是通过远程通信线路提交。提交的作业由系统外存收容成为后备作业。

(2) 成批处理。操作员把用户提交的作业分批进行处理。每批中的作业将由操作系统或监督程序负责作业间自动调度执行。

(3) 多道程序运行。按多道程序设计的调度原则，从一批后备作业中选取多道作业调入内存并组织它们运行，成为多道批处理。

多道批处理系统的优点是由于系统资源为多个作业所共享，其工作方式是作业之间自动调度执行，并在运行过程中用户不干预自己的作业，从而大大提高了系统资源的利用率和作业吞吐量。其缺点是无交互性，用户一旦提交作业就失去了对其运行的控制能力；而且是批处理的，作业周转时间长，用户使用不方便。

值得一提的是，不要把多道程序系统(multiprogramming)和多重处理系统(multipro-

cessing)相混淆。一般讲,多重处理系统配置多个 CPU,因而能真正同时执行多道程序。当然,要想有效地使用多重处理系统,必须采用多道程序设计技术;反之不然,多道程序设计原则上不一定要求有多重处理系统的支持。多重处理系统比起单处理系统来说,虽增加了硬件设施,却换来了提高系统吞吐量、可靠性、计算能力和并行处理能力等好处。

### 1.3.2 分时系统

分时系统一般采用时间片轮转的方式,使一台计算机为多个终端用户服务,对每个用户能保证足够快的响应时间,并提供交互会话能力。因此它具有下述特点。

(1) 交互性。交互会话工作方式给用户带来了许多好处。第一,用户可以在程序动态运行情况下对其加以控制,从而加快调试过程,提供了软件开发的良好环境。第二,用户上机提交作业方便。特别对于远程终端用户,不必将其作业交给机房,在自己的终端上就可以提交、调试并运行其程序。第三,分时系统还为用户之间进行合作提供方便。他们可以通过文件系统、电子邮件或其他通信机制彼此交换数据和信息,共同完成某项任务。

(2) 多用户同时性。多个用户同时在自己的终端上上机,共享 CPU 和其他资源,充分发挥系统的效率。

(3) 独立性。由于采用时间轮转方式使一台计算机同时为多个终端服务,对于每个用户的操作命令又能快速响应,因此,用户都感觉不到有别人也在使用该台计算机,如同自己独占计算机一样。

分时操作系统是一个联机(on-line)、多用户(multi-user)、交互式(interactive)的操作系统。UNIX 是当今最流行的一种多用户分时操作系统,但 CTSS(Compatible Time Sharing System)和 MUTICS(MULTIplexed Information and Computing Service)这两个系统也是值得一提的。前者是一个实验性的分时系统,在 1963 年由 MIT 研制成功。后者是由 MIT、Bell 实验室和 GE 公司联合在 1965 年开始设计的,尽管最后它并没有取得成功,但对 UNIX 的研制是有影响的。

### 1.3.3 实时系统

实时系统是另外一类联机的操作系统。它主要是随着计算机应用于实时控制和实时信息处理领域中而发展起来的。

实时系统的主要特点是提供即时响应和高可靠性。系统必须保证对实时信息的分析和处理的速度比其进入系统的速度要快,而且系统本身要安全可靠,因为像生产过程的实时控制、武器系统的实时控制、航空订票、银行业务等实时事务系统,信息处理的延误或丢失往往会带来不堪设想的后果。实时系统往往具有一定的专用性,它大多用于嵌入式计算中。与批处理系统和分时系统相比,实时系统的资源利用率可能较低。

设计实时操作系统要考虑这样一些因素:

(1) 实时时钟管理(定时处理和延时处理)。

(2) 连续的人-机对话,这对实时控制往往是必需的。

(3) 过载保护。在实时系统中,进入系统的实时任务的时间和数目有很大的随意性,因而在某一时刻有可能超出系统的处理能力,这就是所谓过载问题,要求采取过载保护措施。例如,对于短期过载,把输入任务按一定的策略在缓冲区排队,等待调度;对于持续性过载,

可能要拒绝某些任务的输入;在实时控制系统中,则应及时处理某些任务,放弃某些任务或降低对某些任务的服务频率。

(4) 高度可靠性和安全性需采取冗余措施。双机系统前后台工作,包括必要的保密措施等。

### 1.3.4 通用操作系统

批处理系统、分时系统和实时系统是操作系统的3种基本类型,在此基础上又发展了具有多种类型操作特征的操作系统,称为通用操作系统。它可以同时兼有批处理、分时、实时处理和多重处理的功能。

### 1.3.5 个人计算机上的操作系统

个人计算机上的操作系统是联机的交互式单用户操作系统,它提供的联机交互功能与通用分时系统所提供的很相似。由于是个人专用,因此在多用户和分时所要求的对处理机调度、存储保护方面将会简单得多。然而,由于个人计算机的应用普及,对于提供更方便友好的用户接口的要求会越来越迫切。

多媒体技术已迅速进入个人计算机系统,多媒体计算机给办公室、家庭和个人提供声、文、图数据并茂的全面的信息服务。它要求计算机具有高速信号处理、大容量的内存和外存、大数据量宽频带传输等能力,能同时处理多个实时事件。要求有一个具有高速数据处理能力的实时多任务操作系统。

目前在个人计算机上使用的操作系统以 Windows 系列和 Linux 为主。

### 1.3.6 网络操作系统

计算机网络是通过通信设施将物理上分散的、具有自治功能的多个计算机系统互联起来的,实现信息交换、资源共享、可互操作和协作处理的系统。它具有以下特征:

(1) 计算机网络是一个互联的计算机系统的群体。这些计算机系统在物理上是分散的,可在一个房间里,在一个单位里,在一个城市或几个城市里,甚至在全国或全球范围内。

(2) 这些计算机是自治的,每台计算机有自己的操作系统,各自独立工作,它们在网络协议控制下协同工作。

(3) 系统互联要通过通信设施(硬件和软件)来实现。

(4) 系统通过通信设施执行信息交换、资源共享、互操作和协作处理,实现多种应用要求。互操作(interoperation 或 interoperability)和协作处理(interworking)是计算机网络应用中更高层次要求的特征。它需要有一个环境支持互联的网络中的异种计算机系统之间的进程通信,实现协同工作和应用集成。

网络操作系统的研制开发是在原来各自的计算机操作系统的基础上进行的,按照网络体系结构的各个协议标准进行开发,包括网络管理、通信、资源共享、系统安全和多种网络应用服务等达到上述诸方面的要求。

由于网络计算的出现和发展,现代操作系统的主要特征之一就是具有上网功能,因此,除了在20世纪90年代初期 Novell 公司的 NetWare 等系统被称为网络操作系统之外,人们一般不再特指某个操作系统为网络操作系统。

### 1.3.7 分布式操作系统

粗看起来，分布式系统与计算机网络系统没有多大区别。分布式系统也可以定义为通过通信网络将物理上分布的、具有自治功能的数据处理系统或计算机系统互联起来，实现信息交换和资源共享，协作完成任务。但是有以下一些明显的区别应予考虑：

(1) 作为计算机网络，现在已制定了明确的通信网络协议体系结构及一系列协议族。无论是广域网(WAN)还是局域网(LAN)，即 ISO/OSI 开放式系统互连体系结构及一系列标准协议(或 IEEE、CCITT 相应的标准等)，计算机网络的开发都遵循协议，而对于各种分布式系统并没有制定标准的协议。当然，计算机网络也可认为是一种分布式系统。

(2) 分布式系统要求一个统一的操作系统，实现系统操作的统一性。为了把数据处理系统的多个通用部件合并成为一个具有整体功能的系统，必须引入一个高级操作系统。各处理机有自己的私有操作系统，必须有一个策略使整个系统融为一体，这就是高级操作系统的任务，它可以采用两种形式，一种形式是在每个处理机的私有操作系统之外独立存在，私有操作系统可以识别和调用它；另一种形式是在各处理机私有操作系统的基础上加以扩展。对于各个物理资源的管理，高级操作系统和各私有操作系统之间，不允许有明显的主从管理关系。

在计算机网络中，实现全网统一管理的网络管理系统已成为越来越重要的组成部分。

(3) 系统的透明性。分布式操作系统负责全系统的资源分配和调度、任务划分、信息传输控制协调工作，并为用户提供一个统一的界面和标准的接口，用户通过这一界面实现所需要的操作和使用系统资源，至于操作定在哪一台计算机上执行或使用哪台计算机的资源则是系统的事，用户是不用知道的，即系统对用户是透明的。但是对计算机网络，若一台计算机上的用户希望使用另一台计算机上的资源，则必须明确指明是哪台计算机。

(4) 分布式系统的基础是网络。它和常规网络一样具有模块性、并行性、自治性和通用性等特点，但它比常规网络又有进一步的发展。因为分布式系统已不仅是一个物理上的松散耦合系统，同时还是一个逻辑上紧密耦合的系统。分布式系统由于更强调分布式计算和处理，因此对于多机合作和系统重构、坚强性和容错能力有更高的要求，希望系统有更短的响应时间、高吞吐量和高可靠性。

(5) 分布式系统还处在研究阶段，目前还没有真正实用的系统。而计算机网络已经在各个领域得到广泛的应用。

20 世纪 90 年代出现的网络计算(network computing)已使分布式系统变得越来越现实。特别是 SUN 公司的 Java 语言和运行在各种通用操作系统之上的 Java 虚拟机和 Java OS 的出现，更进一步加快了这一趋势。另外，软件构件技术的发展也加快了分布式操作系统的实现。

## 1.4 操作系统功能

如前所述，操作系统的职能是管理和控制计算机系统中的所有硬件和软件资源，合理地组织计算机工作流程，并为用户提供一个良好的工作环境和友好的接口。计算机系统的主要硬件资源有处理机、存储器、外存储器和输入输出设备。软件和信息资源往往以文件形式存储在

外存储器。下面从资源管理和用户接口的观点分5个方面来说明操作系统的基本功能。

### 1.4.1 处理机管理

在单道作业或单用户的情况下,处理机为一个作业或一个用户所独占,对处理机的管理十分简单。但在多道程序或多用户的情况下,要组织多个作业同时运行,就要解决处理机分配调度策略、分配实施和资源回收等问题。这就是处理机管理功能。正是由于操作系统对处理机管理策略的不同,其提供的作业处理方式也就不同,例如批处理方式、分时处理方式和实时处理方式。呈现在用户面前,就成为了具有不同性质的操作系统。

### 1.4.2 存储管理

存储管理的主要工作是对存储器进行分配、保护和扩充的管理。

(1) 内存分配。在内存中除了操作系统和其他系统软件外,还要有一个或多个用户程序。如何分配内存,以保证系统及各用户程序的存储区互不冲突,这是内存分配问题。

(2) 存储保护。系统中有多个程序在运行,如何保证一道程序在执行过程中不会有意或无意地破坏另一道程序?如何保证用户程序不会破坏系统程序?这是存储保护要解决的问题。

(3) 内存扩充。当用户作业所需要的内存量超过计算机系统所提供的内存容量时,如何把内部存储器和外部存储器结合起来管理,为用户提供一个容量比实际内存大得多的虚拟存储器,而用户使用这个虚拟存储器和使用内存一样方便,这就是内存扩充所要完成的任务。

### 1.4.3 设备管理

(1) 通道、控制器和输入输出设备的分配和管理。现代计算机常常配置有种类很多的输入输出设备,这些设备具有很不相同的操作性能,特别是它们对信息传输和处理的速度差别很大,并且,它们常常是通过通道控制器与主机发生联系的。设备管理的任务就是根据一定的分配策略,把通道、控制器和输入输出设备分配给请求输入输出操作的程序,并启动设备完成实际的输入输出操作。为了尽可能发挥设备和主机的并行工作能力,常需要采用虚拟技术和缓冲技术。

(2) 设备独立性。输入输出设备种类很多,使用方法各不相同。设备管理应为用户提供一个良好的界面,而不必去涉及具体的设备特性,以使用户能方便、灵活地使用这些设备。

### 1.4.4 信息管理(文件系统管理)

上述3种管理都是针对计算机的硬件资源的管理。信息管理(文件系统管理)则是对系统的软件资源的管理。

我们把程序和数据统称为信息或文件。一个文件在暂时不用时,就被放到外部存储器(如磁盘、磁带或光盘等)上保存起来。这样,外存上保存了大量的文件。对这些文件如不能很好地管理,就会引起混乱,甚至遭到破坏。这就是管理信息文件需要解决的问题。

信息的共享、保密和保护也是文件系统所要解决的。如果系统允许多个用户协同工作,那么就应该允许用户共享信息文件。但这种共享应该是受控制的,应该有授权和保密机制。还要有一定的保护机制以免文件被非授权用户调用和修改,即使在意外情况下,如系统失效

或用户对文件使用不当，也能尽量保护信息免遭破坏。也就是说，系统是安全可靠的。

### 1.4.5　用户接口

上述 4 种管理是操作系统对资源的管理。除此以外，操作系统还为用户提供方便灵活地使用计算机的手段，即提供一个友好的用户接口。一般来说，操作系统提供两种方式的接口来和用户发生关系，为用户服务。

一种用户接口是程序一级的接口，即提供一组广义指令（或称系统调用、程序请求）供用户程序和其他系统程序调用。当这些程序要求进行数据传输、文件操作或有其他资源要求时，通过这些广义指令向操作系统提出申请，并由操作系统代为完成。

另一种接口是作业一级的接口，提供一组控制操作命令（或称作业控制语言，或像 UNIX 中的 Shell 命令语言）供用户去组织和控制自己作业的运行。作业控制方式典型地分两大类：脱机控制和联机控制。操作系统提供脱机控制作业语言和联机控制作业控制语言。

## 1.5　计算机硬件简介

如前所述，操作系统管理和控制计算机系统中所有软硬件资源。同时，因为操作系统是一个运行于硬件之上的系统软件，所以必须对操作系统运行的硬件环境有所了解。本节简要介绍计算机硬件系统。

### 1.5.1　计算机的基本硬件元素

构成计算机的基本硬件元素有 4 种：处理器、存储器、输入输出控制与总线、外部设备。这些基本元素的逻辑关系如图 1.6 所示。

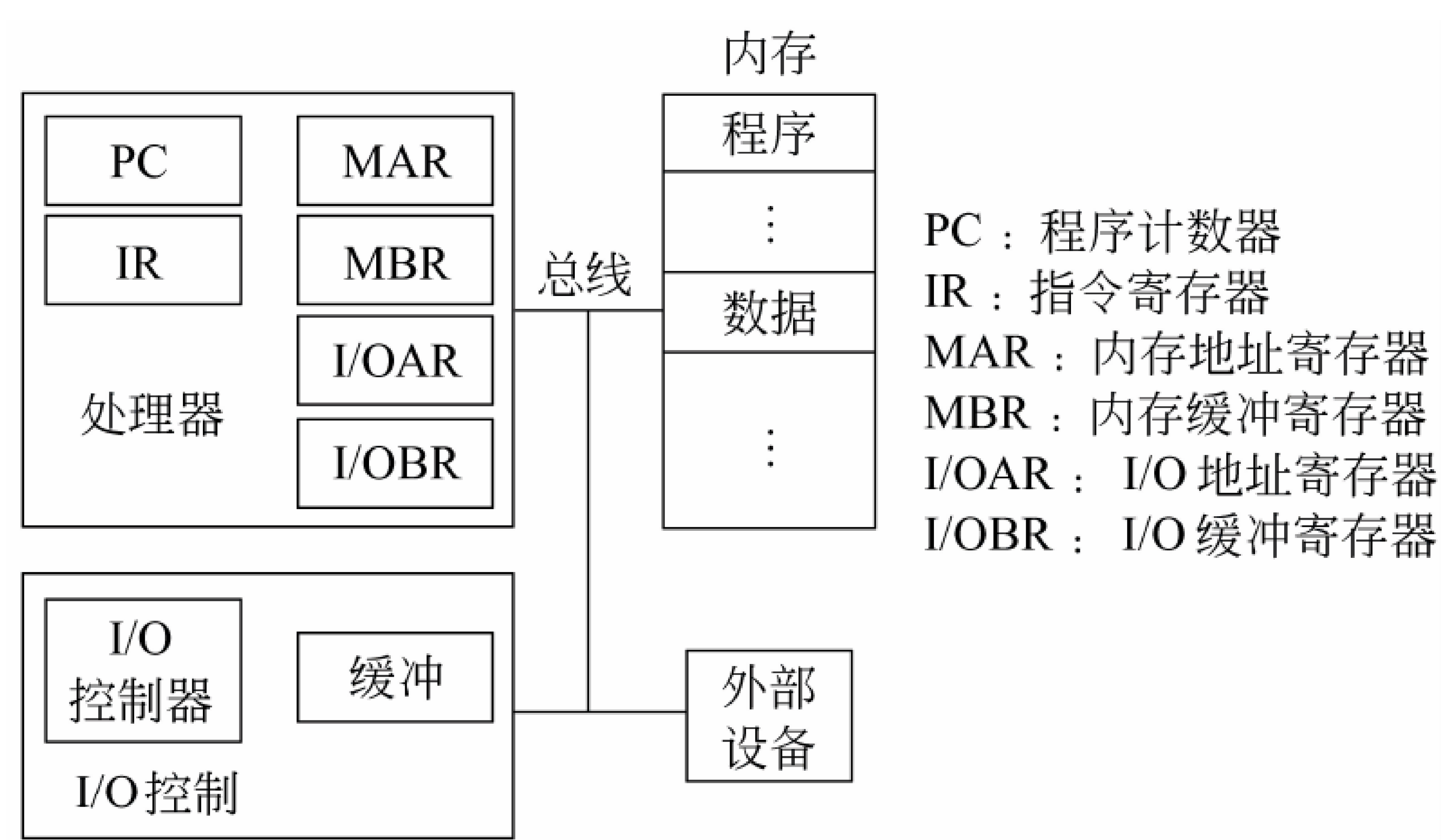



图 1.6　计算机的基本硬件元素

处理器控制和执行计算机的指令操作。一台计算机中可以有多个处理器或单个处理器。多处理器和单处理器的计算机操作系统在设计和功能上都有较大区别，本书主要讨论单处理器的操作系统。单处理器也称 CPU。存储器用来储存数据和程序。存储器可分为内存与外存，以及用于数据和程序暂时存储用的缓冲器与高速缓存(cache)等。

输入输出控制器与缓冲器主要用来控制和暂时存储外部设备与计算机内存之间交换的数据和程序。

外部设备范围很广，它们是获取和输出数据与程序的基本单位，包括数字式设备和模拟式设备。不过，模拟式设备要通过模/数转换后才能把模拟信号输入到计算机，而计算机输出的数字信号则要通过数/模转换之后才能在模拟设备上显示或输出。

计算机系统的各种设备通过总线互相连接。总线是连接计算机各部件的通信线路。计算机系统的总线有单总线和多总线之分。单总线是指处理机、外部设备和存储器等都连接在一起的总线结构，而多总线则指把系统的 CPU 和内存分开连接，外部设备和外存等也用其他总线分开连接进行管理和数据传送的总线结构。显然，不同的总线结构对操作系统的设计和性能有不同的影响。

### 1.5.2 与操作系统相关的几种主要寄存器

寄存器与操作系统密切相关，是在处理机中交换数据的速度比内存更快、体积也更小、而价格又更贵的暂存器件。寄存器的功能可分为两类，即用户可编程的寄存器以及控制与状态寄存器。机器语言或汇编语言的程序员可对用户可编程寄存器进行操作，以获得更高的执行效率等。而控制与状态寄存器则被用来对处理机的优先级、保护模式或用户程序执行时的调用关系等进行控制和操作。

一般来说，用户可编程寄存器和控制与状态寄存器之间没有严格的区分和限制，在不同的系统中，寄存器的功能和作用不完全相同。

**1. 用户可编程寄存器**

典型的用户可编程寄存器包括以下几种。

（1）数据寄存器。编程人员可以通过程序赋予数据寄存器众多的功能。一般来说，对数据进行操作的任何机器指令都被允许访问数据寄存器。不过，根据硬件设置的规定，这些寄存器也可能只被允许进行浮点运算或被其他某些规定所限制。

（2）地址寄存器。一般用来存放内存中某个数据或指令的地址，或者存放某段数据与指令的入口地址以及被用来进行更复杂的地址计算。下面几种寄存器都可被认为是地址寄存器：

① 地址标识位寄存器；

② 内存管理用各种始地址寄存器；

③ 堆栈指针；

④ 设备地址寄存器等。

（3）条件码寄存器。也称标志寄存器，其比特位由处理机硬件设置。例如，一次算术运算可能导致条件码寄存器被设置为正、负、零或溢出。

**2. 控制与状态寄存器**

典型的控制与状态寄存器包括以下几种：

（1）程序计数器（PC）。该计数器内装有下一周期被执行指令的地址。

（2）指令寄存器（IR）。该寄存器内装有待执行的指令。

（3）程序状态字（PSW）寄存器。该寄存器的各个比特位代表系统中当前的各种不同状态与信息，例如执行模式是否允许中断等。

（4）中断现场保护寄存器。如果系统允许不同类型的中断存在，则会设置一组中断现场保护寄存器以便保存被中断程序的现场和链接中断恢复处。

（5）过程调用堆栈。用来存放过程调用时的调用名、调用参数以及返回地址等。

寄存器被广泛应用于计算机系统中，它们与操作系统有着非常直接和密切的关系。事实上，操作系统设计人员只有在完全掌握和了解硬件厂商所提供的各种寄存器的功能和接口之后，才能进行操作系统设计。

### 1.5.3 存储器的访问速度

硬件厂商提供了不同种类的存储器件，这些存储器件包括以下两类：可移动存储介质（例如光盘、磁盘和磁带等）、硬盘、磁盘缓存（disk cache）、内存、高速缓存以及寄存器等。设计人员应对各种存储器件的访问速度和性能等有充分了解，才能在设计中以最好的性能价格比设计存储管理系统。

一般来说，容量越大的存储介质，访问速度会越慢，但单位存储的成本越低，例如光盘和磁盘。反过来说，如果存储介质的访问速度越高，则它的成本也会越高，例如寄存器。

存储器件的访问速度与存储量的关系如图 1.7 所示。

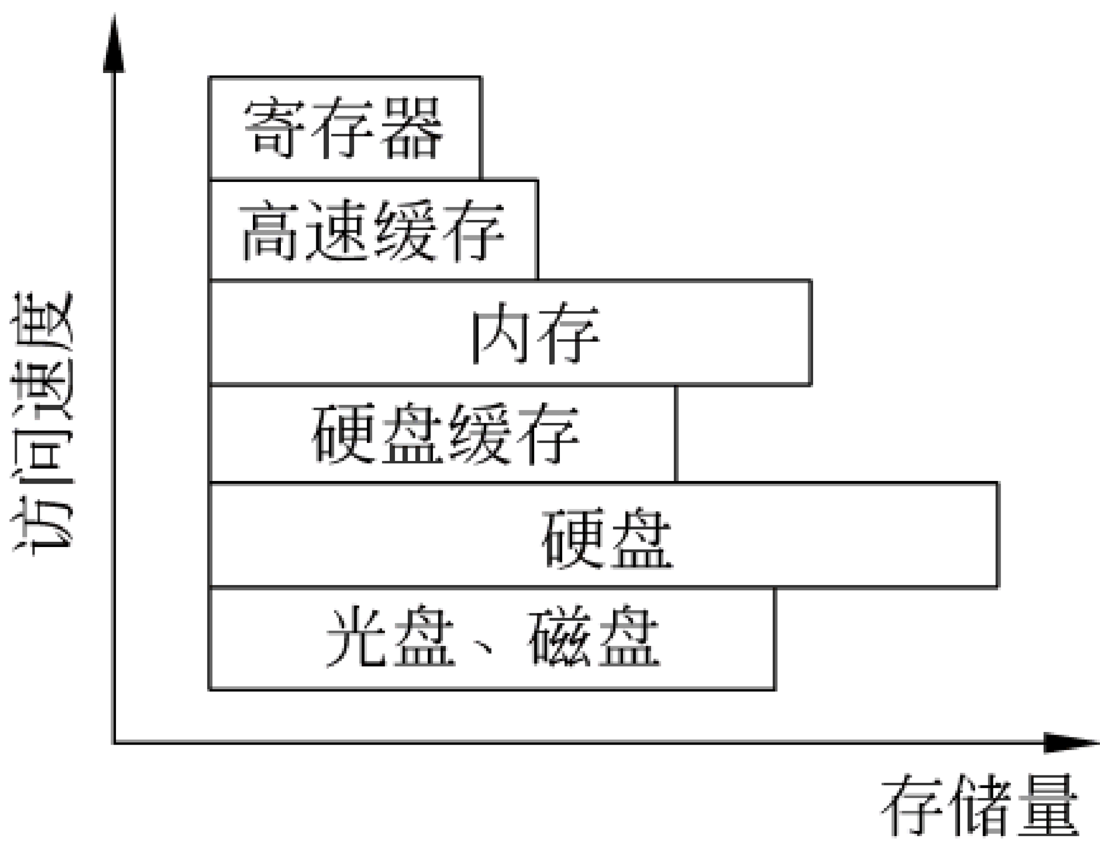


图 1.7 存储介质的访问速度

除了上述的寄存器与存储介质之外，与操作系统设计相关的硬件器件还有中断机构和输入输出设备控制部分，例如通道和 DMA 器件等，这些部分将在后面相关章节中介绍。

### 1.5.4 指令的执行与中断

计算机提供的最基本功能是执行指令。任何应用程序都只有通过指令的执行才能得以完成。执行指令的基本过程分为两步，即处理机从内存读入指令的过程和指令执行的过程。其中，读指令是根据程序计数器（PC）所指的地址读入，而执行的指令则是指令寄存器（IR）中的指令。

指令的读入和执行过程称为一个执行周期。一个指令的基本执行周期如图 1.8 所示。

指令的执行涉及处理机与内存之间的数据传输，或者是处理机与外部设备之间的数据传输等。指令的执行也涉及数据处理，例如算术运算或逻辑运算。另外，指令的执行还可以是对其他指令的控制过程。

一条指令的执行可以是上述几种情况的组合。

另外，在指令的执行过程中或一条指令执行结束时，尽管指令地址计数器中已指明了下一条被访问指令的地址，但是，外部设备或计算机内部可能会发来亟须处理的数据或其他紧急事件处理信号。这就需要处理机暂停正在执行的程序，转去处理相应的紧急事件，待处理完毕后再返回原处继续执行，这一过程称为中断，如图 1.9 所示。

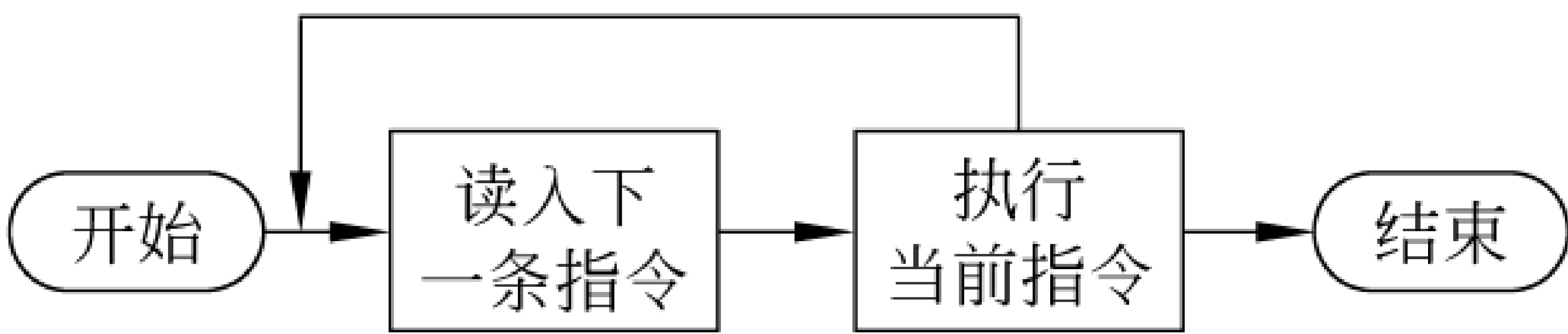


图 1.8 指令的执行周期

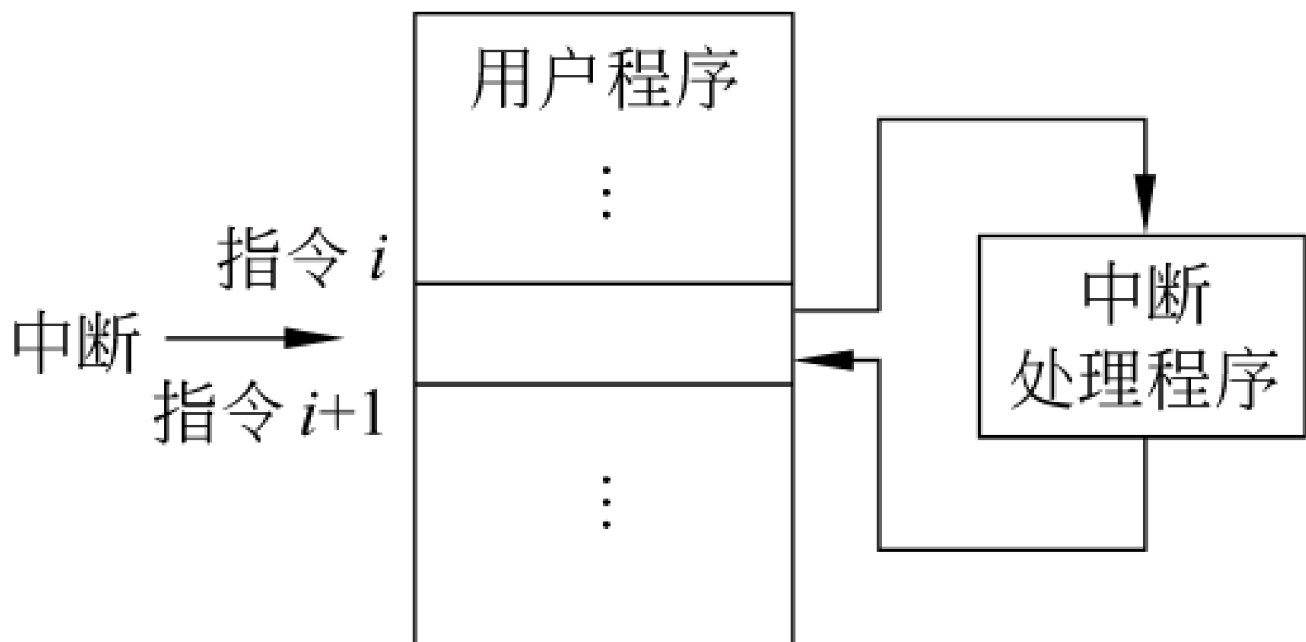


图 1.9 中断的执行过程

中断给操作系统设计带来许多好处，首先使得实时处理许多紧急事件成为可能；其次，中断可以增加处理机的执行效率；另外，中断还可以简化操作系统的程序设计。

具有中断处理时的指令执行过程如图 1.10 所示。

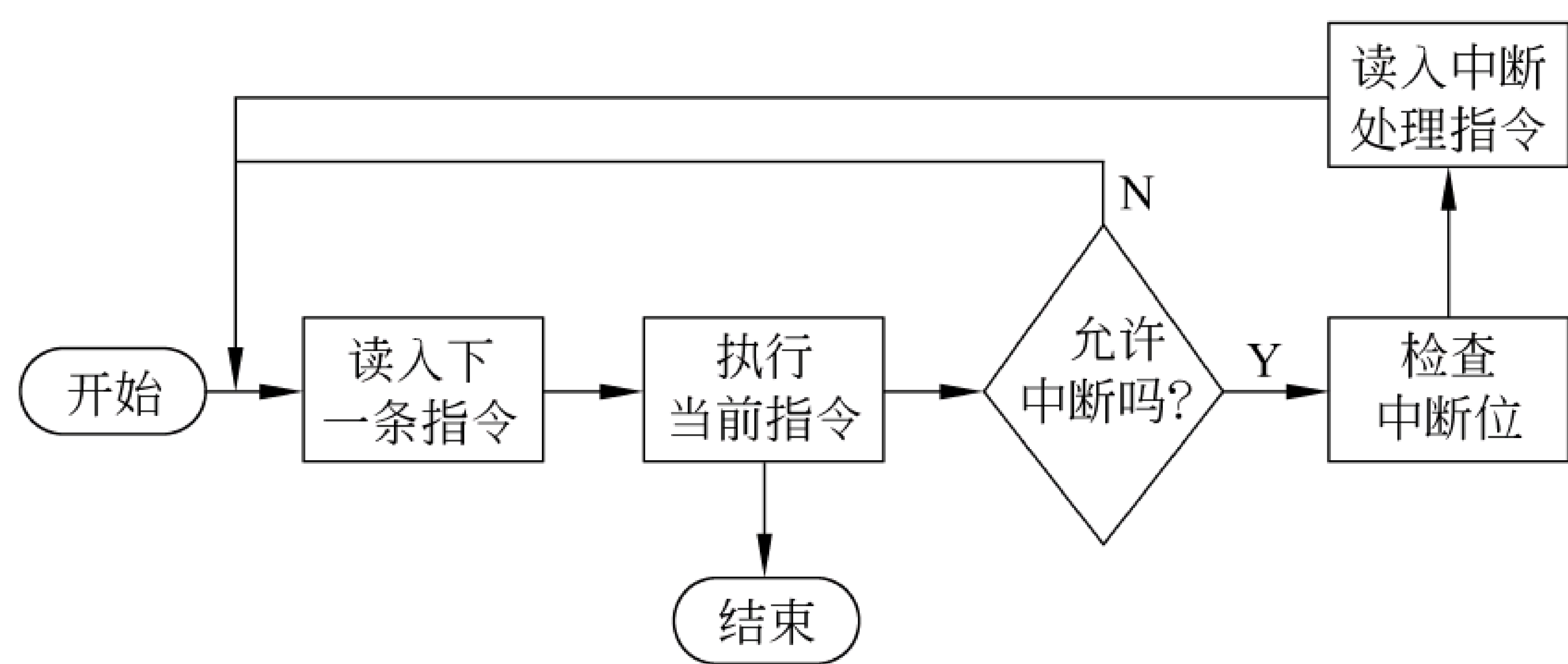



图 1.10　中断处理时的指令执行过程

系统发生中断时，处理机收到中断信号，从而不能继续执行程序计数器中所指的原程序。这时处理机将保存当前的执行现场（也就是各寄存器中的值）并调用新的程序到处理机上执行。

### 1.5.5　操作系统的启动

操作系统负责管理计算机软硬件资源。可是，操作系统本身也是一种资源。在计算机系统中，操作系统是如何启动运行的呢？

事实上，当用户启动计算机的电源时，计算机硬件会自动产生一个中断信号，这个中断信号触发计算机处理器（CPU）中的一段指令执行。该段指令的执行结果将是发现外部存储设备中操作系统引导区（boot block）的位置。如果计算机的外部存储设备中已经安装了操作系统，则操作系统引导区中的代码将被自动导入计算机的内存，并开始执行。引导区代码的执行结果是将操作系统程序加载到计算机内存中的指定区域，并初始化计算机的有关硬件，例如寄存器、终端设备，以及各种计算机运行所需要的数据结构等。至此，操作系统程序开始启动，并为用户提供相应的用户界面，开始提供用户所需要的各种服务。

## 1.6　算法的描述

操作系统设计和原理描述中涉及许多算法。一般来说，这些算法可以用自然语言或流程图方式描述。有许多书中也用类 Pascal 语言或其他形式描述语言来描述算法。为了描述简单起见，本书定义下述关键词描述算法中的有关过程。

```
begin
end
```

分别表示算法的开头和结束。

```
repeat
    操作
until 条件
```

表示当条件未被满足时重复所描述的操作。

```
while 条件
do
    操作
od
```

表示当条件满足时,进行相应的操作。关键词 do 和 od 分别表示操作的开始和结束。

```
if 条件
then
    操作
else
    操作
fi
```

表示满足 if 所指定的条件时,进行 then 后的相关操作,否则完成 else 后的相关操作。关键词 fi 表示条件判断的结束。

例如,图 1.8 所示的指令执行周期可被描述为

```
repeat  IR←M[PC];
        PC←PC+1;
        execute[IR];
until   CPU halt;
```

其中,M[PC]表示地址为 PC 的内存单元中的指令内容。

另一个例子如下:

令 $p[1:n]$为 1 到 $n(n>1)$的整数置换。

设 $i=1,2,3,4,5,6,7$;

$p[i]=4,7,3,2,1,5,6$;

描述 $p[i]$的巡回置换算法。(巡回置换指 $k\in[1:n]$时,$k=p[p[\cdots p[k]\cdots]]$的置换。)

**解:**

```
begin
    local    x,k;
    k←1;
    while    k<=7    do
        x←k;
        repeat    print(x);
                  x←p[x];
        until     x=k;
        k←k+1;
    od
end
```

## 1.7 研究操作系统的几种观点

上面各节讨论了几种操作系统的基本概念、操作系统发展的历史、操作系统的分类和功能以及操作系统所依赖的硬件基础等问题,使我们认识到,操作系统是计算机资源有效使用

的管理者和为用户提供的友好接口。这实质上代表了讨论操作系统的一种观点。

本节简单地讨论操作系统研究中的不同观点，这些观点彼此并不矛盾，只不过代表了对同一事物(操作系统)站在不同的角度来看待的结论。每一种观点都有助于理解、分析和设计操作系统。

### 1.7.1 计算机资源管理者的观点

前面已经指出，操作系统就是指用来管理和控制计算机系统软硬资源的程序的集合，因此它提供了处理机管理、存储管理、设备管理和信息文件管理等功能。对于每种资源的管理都可以从资源情况记录、资源分配策略、资源分配和资源回收等几个方面来加以讨论。

### 1.7.2 用户界面的观点

对于用户来说，对操作系统的内部结构并没有多大的兴趣，他们最关心的是如何利用操作系统提供的服务来有效地使用计算机。因此操作系统提供了什么样的用户界面成为关键问题，即 1.4.5 节中所提出的程序一级和作业一级的两种接口。

### 1.7.3 进程管理的观点

上述两种实际上是静态的观点，没有揭示一个程序在系统中运行的本质过程和管理资源的各种子程序存在的关系。实质上操作系统调用当前程序运行是一个动态过程，特别是现代操作系统的一个重要特征是并发性。并发性是指操作系统控制很多能并发执行的程序段。当然这些并发执行的程序在多处理机系统中可能是真正并行执行的，但在单处理机情况下则是宏观并行、微观顺序执行的。它们可以是完全独立地运行的，也可能以间接或直接方式互相依赖和制约。间接式制约是指并发程序段竞争同一资源，获得者执行，未获得者挂起，等待资源；直接式制约是指一个程序段等待另一程序段执行的信息结果后才能执行。因此并发的程序段不仅会受到其他程序段活动的制约，也会受到系统资源分配情况的制约。一个程序段可能在运行，也可能因等待某些资源或信息处于挂起状态等。因此只用“操作系统是资源管理程序”这一概念不能揭示它们在系统中活动联系及其状态变化，从而引入“进程”(process)的概念，有时也称为“任务”(task)或“活动”(active)。所谓“进程”是指并发程序的执行。

用进程观点来研究操作系统就是围绕进程运行过程(即并发程序执行过程)来讨论操作系统，就能讨论清楚“这些资源管理程序在系统中进行活动的过程”，对操作系统功能就能获得更多的认识。

## 本 章 小 结

计算机系统由硬件和软件组成。操作系统是计算机系统中的系统软件，是管理和控制计算机硬件和软件资源，合理组织计算机工作流程，以便有效利用这些资源为用户提供一个具有足够的功能、使用方便、可扩展、安全和可管理的工作环境，从而在计算机与其用户之间起到接口的作用。

随着计算机的发展，操作系统经历了从手工操作到通用操作系统的发展历程。批处理

操作系统的主要特征是用户可脱机使用计算机、成批处理及多道程序运行，其优点是共享系统资源、系统资源利用率和作业吞吐量高，缺点是无交互性、作业周转时间长。分时系统采用时间片轮转方式使一台计算机为多个终端用户服务，具有交互性、多用户同时性和独立性等特征。实时系统应用于实时控制和实时信息处理领域，完成对信息及时地分析和处理，其主要特点是即时响应和高可靠性。通用操作系统可以同时兼有批处理、分时、实时处理和多重处理的功能，或其中两种以上的功能。个人计算机上的操作系统是联机的交互式单用户操作系统，随着多媒体技术的应用，个人计算机操作系统越来越要求成为具有高速数据处理能力的实时多任务操作系统。网络操作系统是通过计算机网络将多个计算机系统互联，实现信息交换、资源共享、可互操作和协作处理的系统。分布式操作系统是通过通信网络将物理上分布的具有自治功能的数据处理系统或计算机系统互联、实现信息交换和资源共享、协作完成任务的系统。以上这些都是操作系统的基本类型。

操作系统的基本功能包括处理机管理、存储管理、设备管理、信息管理以及用户接口。这些基本功能的目标是管理和控制计算机系统所有软硬件资源，为用户提供一个良好的工作环境和友好的接口。计算机最基本的功能是执行指令，指令的执行涉及处理机与内存之间的数据传输或处理机与外部设备之间的数据传输。指令执行过程中，若外部设备或计算机内部发来亟须处理的数据或其他紧急事件处理信号时，系统将进入中断处理过程。中断时处理机将保存当前的执行现场，并调用新的程序到处理机上执行。

对操作系统的研究有不同的观点，如操作系统是计算机资源管理者的观点、用户界面的观点以及进程管理的观点等，这些观点代表对操作系统不同角度的看法，都有助于理解、分析和设计操作系统。

## 习　　题

1.1　什么是操作系统的基本功能？

1.2　什么是批处理、分时和实时系统？它们各有什么特征？

1.3　多道程序设计(multiprogramming)和多重处理(multiprocessing)有何区别？

1.4　讨论操作系统可以从哪些角度出发，如何把它们统一起来。

1.5　写出1.6节中巡回置换算法的执行结果。

1.6　设计计算机操作系统时与哪些硬件器件有关？

# 第2章 操作系统用户界面

本章主要从用户使用和系统管理两方面出发,讨论操作系统为用户提供的编程接口和命令控制接口。首先讨论操作系统的管理概念,然后介绍系统调用与编程接口,最后介绍操作系统用户界面示例。

## 2.1 简 介

用户界面是操作系统的重要组成部分。用户界面负责用户和操作系统之间的交互。即用户通过用户界面向计算机系统提交服务需求,计算机通过用户界面向用户提供用户所需要的服务。

一般来说,计算机系统的用户有两类。

一类是使用和管理计算机应用程序的用户,也就是被服务者。这类用户又可进一步分为普通用户和管理员用户。其中普通用户只是使用计算机的应用服务,以解决实际的应用问题,例如事务处理、过程控制等。管理员用户则负责计算机和操作系统的正常与安全运行。

另一类用户是程序开发人员。程序开发人员需要使用操作系统所提供的编程功能开发新的应用程序,完成用户所要求的服务。

操作系统为普通用户、管理员用户以及编程人员提供不同的用户界面。

操作系统为普通用户和管理员用户提供的界面由一组以不同形式表示的操作命令组成。其中,每个命令实现和完成用户所要求的特定功能和服务,例如上网、在线处理和办公处理等。

不同计算机操作系统为用户提供的用户操作命令和表现形式不同,不同时期的操作系统为用户提供的操作命令和表现形式也不同。例如,Windows 系统、UNIX 和 Linux 提供给用户的操作命令都是不同的。

再者,同一操作系统为普通用户与管理员用户提供的命令集合也是不一样的。读者可以自己考察 UNIX 系统或 Linux 系统中管理员用命令集与普通用户命令集的区别。

操作系统的操作命令界面称为命令控制界面。

另外,操作系统为编程人员提供的界面是系统调用。系统调用是操作系统为编程人员提供的唯一界面。不同的操作系统为编程人员提供的系统调用不同。

综上所述,针对不同的用户,操作系统提供不同的用户界面,其中普通用户和管理员用户的界面是一组不同操作命令的集合,它们分别实现用户所要求的不同功能,为用户提供相应的服务;对编程人员提供的是一组系统调用的集合,这些系统调用允许编程人员使用操作系统和程序,开发能够满足用户服务需求的新的控制命令。

## 2.2　一般用户的输入输出界面

要了解计算机是怎样和用户交互的，就必须了解用户怎样使用计算机提供的各种命令以及学会怎样把编制的应用程序变成普通用户可以使用的命令。

在操作系统中，作业(job)或任务(task)是一个常见的概念。在PC等微机系统中，作业的概念有助于人们对问题的认识和管理。

### 2.2.1　作业的定义

回忆一下一般的编程过程，编制一个应用程序大致要经过图2.1所描述的步骤。即由概念或构思出发，经过功能设计、结构设计以及详细设计过程之后，再编制程序和进行编辑输入、编译链接和反复调试之后再形成执行代码并执行，然后输出执行结果和建立相应的文档等。

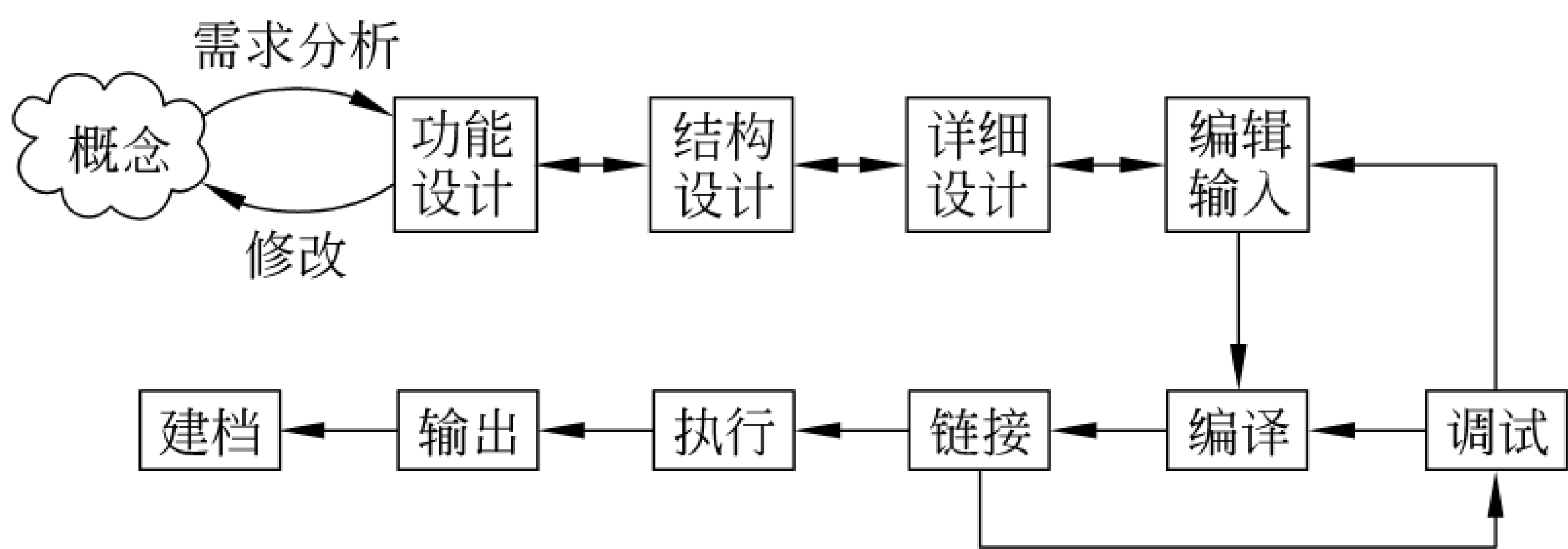


图2.1　一般的编程过程

在图2.1中，直到编辑为止的各步都认为是由人工独立完成的(尽管也有许多支撑软件存在)，但从编辑输入开始的以下各步却是在用户的控制下由计算机完成。

在一次应用业务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次业务处理的全部工作称为一个作业。作业由不同的顺序相连的作业步组成。作业步是在一个作业的处理过程中计算机所做的相对独立的工作。一般来说，每一个作业步产生下一个作业步的输入文件。例如，在图2.1中，编辑输入是一个作业步，它产生源程序文件；编译也是一个作业步，它产生目标代码文件。

从系统的角度看，作业则是一个比程序更广的概念，它由程序、数据和作业说明书组成。系统通过作业说明书控制文件形式的程序和数据，使之执行和操作。而且，在批处理系统中，作业是抢占内存的基本单位，也就是说，批处理系统以作业为单位把程序和数据调入内存以便执行。

需要说明的是，作业的概念一般用于早期批处理系统和现在的大型机、巨型机系统中，对于广为流行的微机和工作站系统，一般不使用作业的概念。

### 2.2.2　作业组织

如上所述，作业由3部分组成，即程序、数据和作业说明书。一个作业可以包含多个程序和多个数据集，但必须至少包含一个程序，否则将不成为作业。作业中包含的程序和数据完成用户所要求的业务处理工作。作业说明书则体现用户的控制意图，由作业说明书在系

统中生成一个称为作业控制块(Job Control Block,JCB)的表格。该表格登记该作业所要求的资源情况、预计执行时间和执行优先级等。从而,操作系统通过该表了解到作业要求,并分配资源和控制作业中程序和数据的编译、链接、装入和执行等。

作业说明书主要包含3方面内容,即作业的基本描述、作业控制描述和资源要求描述。作业基本描述包括用户名、作业名、使用的编程语言名以及允许的最大处理时间等。而作业控制描述则大致包括作业在执行过程中的控制方式,例如是脱机控制还是联机控制,各作业步的操作顺序以及作业不能正常执行时的处理等。资源要求描述包括要求内存大小、外设种类和台数、处理机优先级、所需处理时间、所需库函数或实用程序等。作业说明书的主要内容如图2.2所示。

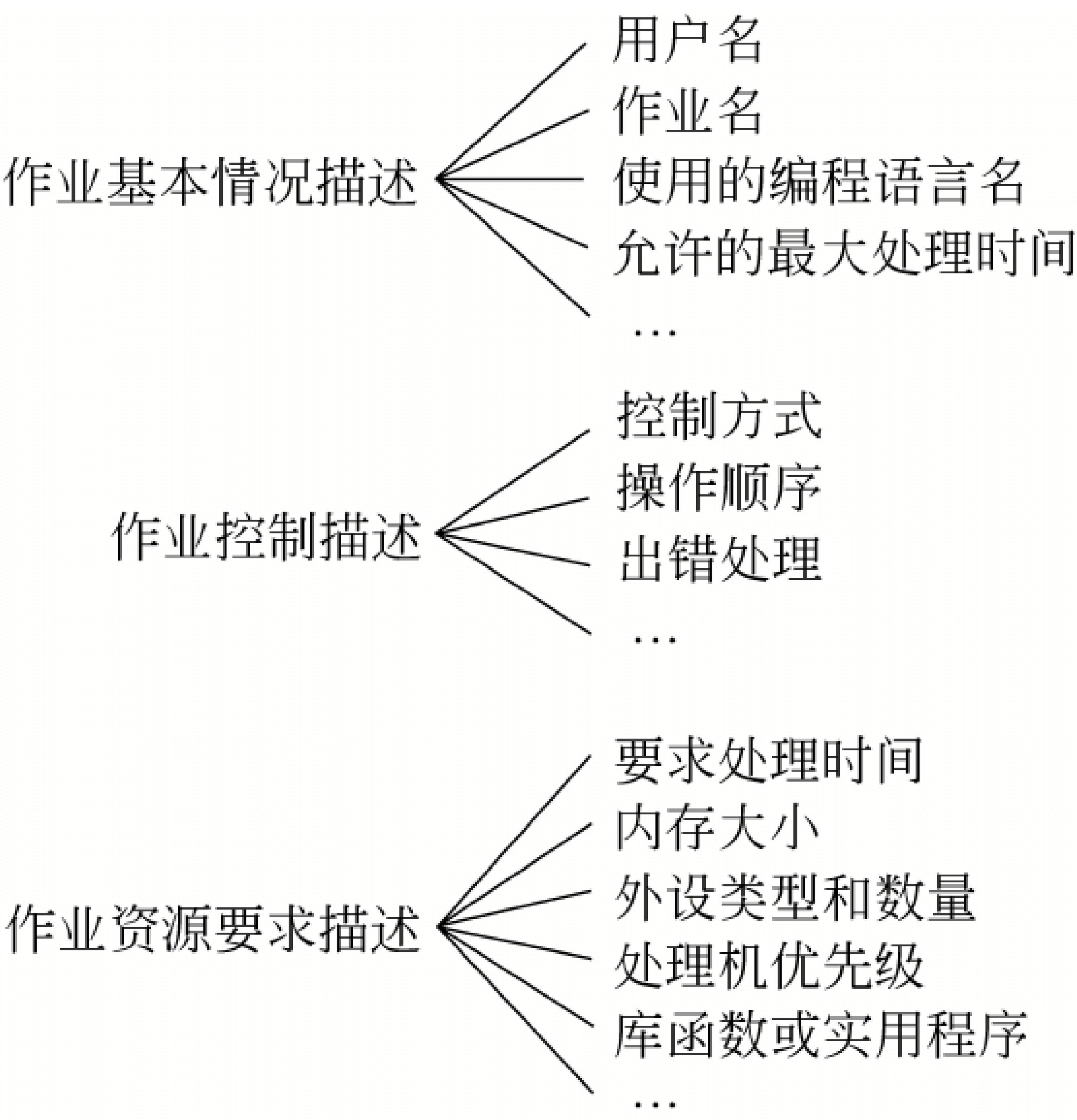


图2.2 作业说明书的主要内容

一般来说,作业说明书方式主要用在批处理系统中,且各计算机厂家都对自己的系统定义有各自的作业说明书的格式和内容。因此,上述作业说明书的内容因计算机厂家而异。不过,无论何种作业说明书,都根据系统提供的控制命令和有关参数按照一定的格式进行编写。

另外,在微机系统和工作站系统中,常用批处理文件或shell程序方式编写作业说明书。

### 2.2.3 一般用户的输入输出方式

输入输出方式可分为5种,即联机输入输出方式、脱机输入输出方式、直接耦合方式、spooling(simultaneous peripheral operations on-line)系统和网络联机方式。

**1. 联机输入输出方式**

联机输入方式大多用在交互式系统中,用户和系统通过交互会话来输入输出作业。在联机方式中,外围设备直接和主机相连接,一台主机可以连接一台或多台外围设备,这些设备可以是键盘、鼠标、显示器或光电笔、打印机等。

**2. 脱机输入输出方式**

脱机输入又称为预输入方式。脱机输入输出方式主要是为了解决设备联机输入输出时速度太慢的问题。脱机输入输出方式利用个人计算机作为外围处理机进行输入输出处理。在个人机上,用户通过联机方式把数据或程序首先输入到后援存储器上,例如优盘等;然后,用户把装有输入数据的后援存储器拿到主机的高速外围设备上和主机连接,从而在较短的时间内完成作业的输入工作。输出过程则与此相反。

**3. 直接耦合方式**

直接耦合方式保留了脱机输入输出方式快速输入的优点,又没有脱机输入输出方式人工干预的缺点,直接耦合方式把主机和外围机通过一个公用的大容量外存直接耦合起来,从而省去了在脱机输入中依靠人工干预来传递后援存储器的过程。在直接耦合方式中,慢速的输入输出过程仍由外围机自己管理,而对公用存储器中的大量数据的高速读写则由

主机完成。直接耦合方式的原理如图 2.3 所示。

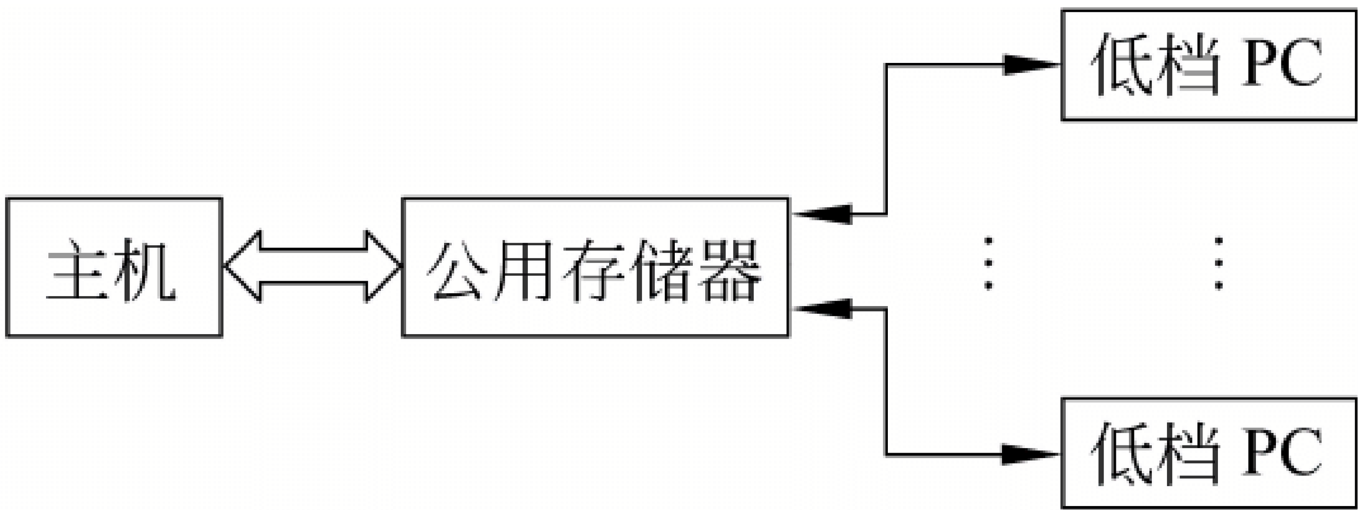



图 2.3　直接耦合方式

直接耦合方式需要一个大容量的公用存储器，把多台外围机、主机和公用存储器固定连接起来。

**4. spooling 系统**

spooling 又可译作外围设备同时联机操作。spooling 系统的工作原理如图 2.4 所示。

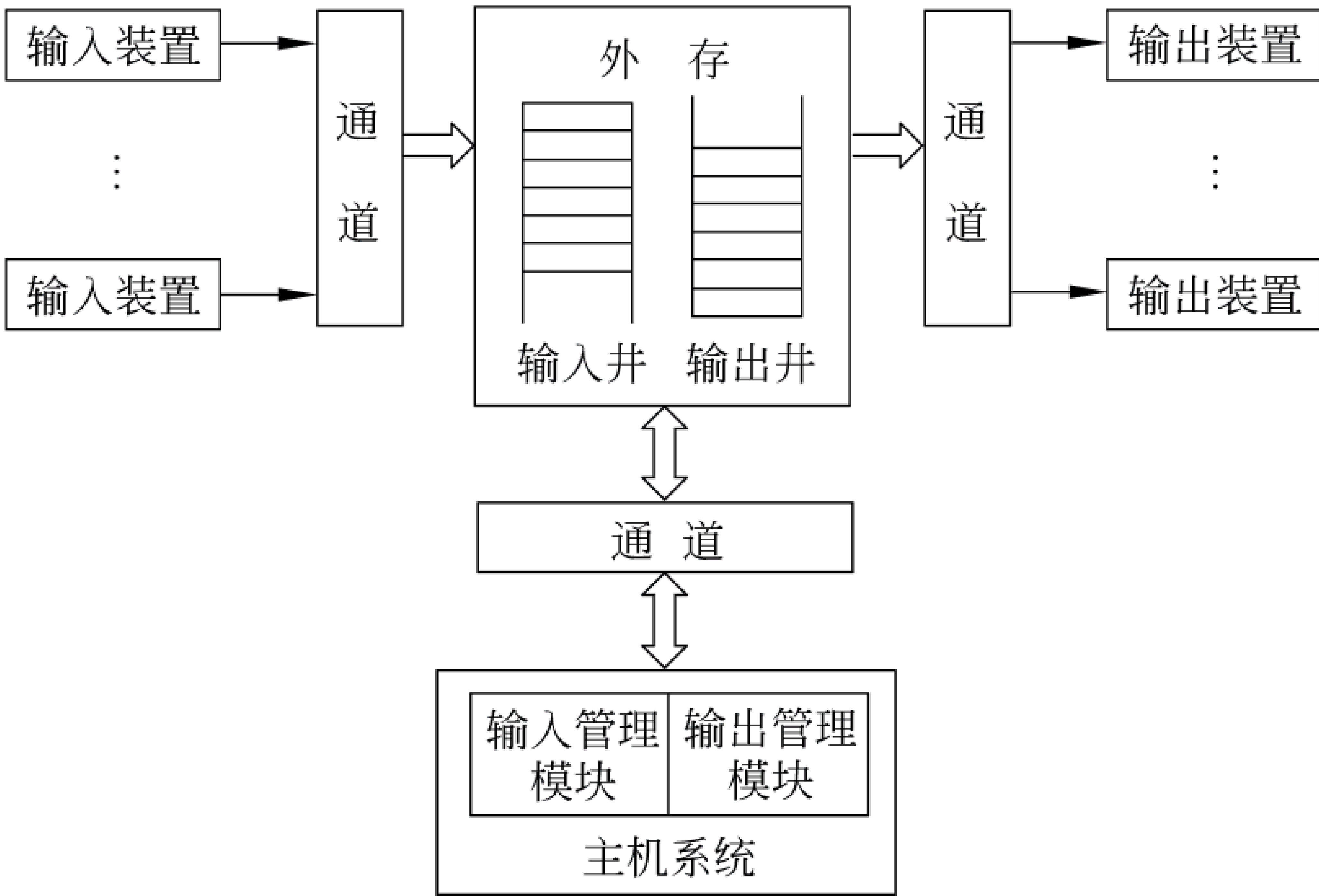



图 2.4　spooling 系统

在 spooling 系统中，多台外围设备通过通道或 DMA 器件和主机与外存连接起来。作业的输入输出过程由主机中的操作系统控制。操作系统中的输入程序包含两个独立的过程，一个过程负责从外部设备把信息读入缓冲区；另一个是写过程，负责把缓冲区的信息送到外存输入井中。这里，外围设备既可以是各种终端，也可以是其他的输入设备，例如纸带输入机或读卡机等。

有关通道和 DMA 的概念，已在计算机原理课中讲述了。通道是一个独立于 CPU 的专管输入输出的处理机，它控制外设或外存与内存之间的信息交换。它有自己的通道指令，以驱动外设进行读写操作。不过，这些指令需要 CPU 执行相应的“启动通道”指令发来启动信号之后才开始执行。DMA 方式类似于通道方式。与通道不同的是，在 DMA 方式中，信息的传送方向、信息传送的源地址和目的地址以及传送长度等都是由 CPU 控制而不是由 DMA 器件控制的。有关通道和 DMA 控制方式，还将在第 9 章中讨论。

spooling 系统的输入方式既不同于脱机方式，也不同于直接耦合方式。在系统输入模块收到作业输入请求信号后，输入管理模块中的读过程负责将信息从输入装置读入缓冲区。当缓冲区满时，由写过程将信息从缓冲区写到外存输入井中。读过程和写过程反复循环，直到一个作业输入完毕。当读过程读到一个硬件结束标志之后，系统再次驱动写过程把最后

一批信息写入外存并调用中断处理程序结束该次输入。然后，系统为该作业建立作业控制块(JCB)，从而使输入井中的作业进入作业等待队列，等待作业调度程序选中后进入内存。

**5. 网络联机方式**

网络联机方式以上述几种输入输出方式为基础。当用户通过计算机网络中的某一台设备对计算机网络中的另一台主机进行输入输出操作时，就构成了网络联机方式。

## 2.3 命令控制界面

如前所述，操作系统为用户提供两个接口。一个是系统为用户提供的各种命令接口，用户利用这些操作命令来组织和控制作业的执行或管理计算机系统；另一个接口是系统调用，编程人员使用系统调用来请求操作系统提供服务，例如申请和释放外设等类资源、控制程序的执行速度等。操作系统的命令控制界面就是用来组织和控制作业运行的。

使用操作命令进行作业控制的主要方式有两种，即脱机方式和联机方式。所谓脱机方式，即用户将作业的执行顺序和出错处理方法一并以作业控制说明书或命令文件的方式提交给系统，由系统按照作业说明书或命令文件中所规定的顺序控制作业执行。在执行过程中，用户无法干涉，只能等待作业正常执行结束或出错停止之后查看执行结果或出错信息，以便修改作业内容或控制过程。

脱机控制方式利用作业控制语言来编写表示用户控制意图的作业控制程序，也就是作业说明书。作业控制语言的语句就是作业控制命令。不同的批处理系统提供不同的作业控制语言。

联机控制方式不同于脱机控制方式，它不要求用户填写作业说明书，系统只为用户提供一组通过键盘或其他操作方式输入的命令。用户使用系统提供的操作命令和系统会话，交互地控制程序执行和管理计算机系统。其工作过程是，用户在系统给出的提示符下输入特定的命令，系统在执行完该命令后向用户报告执行结果；然后，用户决定下一步的操作。如此反复，直到作业执行结束。凡是使用过 DOS、Windows 或 Linux 系统的读者，对联机控制方式都应该是不陌生的。

与脱机控制方式相比，联机控制方式的命令种类要丰富得多。这些命令可大致分为以下几类：

(1) 环境设置。这些命令用来改变终端用户的所在位置、执行路径等。

(2) 执行权限管理。这些命令用来控制用户访问系统和读、写、执行有关文件的权限。例如，用户只有在其口令经过系统核准之后才能进入系统。

(3) 系统管理。该类命令主要用于系统维护、开机与关机、增加或减少终端用户、计时收费等。该类命令是操作系统提供的最为丰富的一类命令，且其中的很大一部分为系统管理员使用。

(4) 文件管理。该类命令用于管理和控制终端用户的文件。例如，复制、移动或删除某个文件或显示文件内容和改变文件名字，以及搜索文件中的特定行或字符等。

(5) 编辑、编译、链接装配和执行编辑命令用于帮助用户输入用户文件，不同的编辑器具有不同的命令集合。

这些命令用于增加、删除输入字符或字符行，用于进行插入、移动甚至绘图等。编译和

链接装配命令则把用户输入的源程序文件编译成目标代码文件之后，再链接成可执行代码文件。执行命令则将链接后的可执行代码文件送入内存启动执行。

（6）通信。通信类命令在单机系统中用于进行主机和远程终端之间的呼叫、连接以及断开等，从而在主机和终端之间建立会话信道。在网络系统中，通信命令除了用于进行有关信道的呼叫、连接和断开等之外，还进行主机和主机之间的信息发送与接收、显示、编辑等工作。

（7）资源要求。用户使用该类命令向系统申请资源，例如申请某台外围设备等。

联机控制方式使用户直接参与控制作业执行，因而大大地方便了用户。但是，在某些情况下，用户反复输入众多的命令也会感到非常烦琐或浪费了许多不必要的时间。例如，在对某个源代码文件进行编译调试之后，需要重新和多个目标代码文件链接。如果这个调试和链接不是一次成功的话（很多情况下是不可能的），那么，用户的控制过程将会非常单调和烦琐。显然，在这种情况下，批处理方式要优于联机控制方式。因此，在现代操作系统中，大都提供批处理方式和联机控制方式。这里，批处理方式既指传统的作业控制语言编写的作业说明书方式，也指那些把不同的交互命令按一定格式组合后的命令文件方式。

近年来，命令控制界面的人机交互方式发生了革命性变化。一个操作系统命令控制界面的好坏成了决定该系统是否能受到欢迎的重要因素。例如，无论是 Windows 系列还是 Linux 系列的操作系统，它们的命令控制界面都是由多窗口的按钮式图形界面组成的。在这些系统中，命令已被开发成一条条能用鼠标点击而执行的简单菜单或小巧图标。而且，用户也可以在提示符的提示下用普通字符方式输入各种命令。最近，用声音控制的命令控制界面也已逐步被开发出来。可以预计，计算机系统的命令控制界面将会越来越方便和越来越拟人化。

## 2.4　Linux 与 Windows 的命令控制界面

现代操作系统的命令控制界面都在朝着多媒体的拟人化方向发展，即一般用户的输入输出界面都在朝着人类自身的交流方式逼近。Linux 和 Windows 的用户界面就是这些系统中最具代表性的两种。

### 2.4.1　Linux 的命令控制界面

Linux 的最大特点是其源代码的免费和开放，而且为普通用户与程序员提供通用的标准接口与界面。

Linux 的命令控制都是用图形化的窗口系统以及 Shell 程序进行的。

Linux 的图形化窗口系统是 X Window。图 2.5 是一个 Ubuntu 系统的窗口界面示例，这是基于 Linux 2.6 内核版本的操作系统。读者也可以从装有 Linux 操作系统的个人计算机中找到其他 Linux 窗口系统用户界面的例子。另外，也可以从网站 www.kde.org 和 www.gnome.org 中找到 Linux 用户界面的相关知识。

一般来说，Linux 命令主要包括以下几类：

（1）系统维护及管理命令，例如 date、setenv 等。

（2）文件操作及管理命令，例如 ls、find 等。




图 2.5　Ubuntu 系统的窗口界面示例

(3) 进程管理命令，例如 kill、at 等。

(4) 磁盘及设备管理命令，例如 df、du、mount 等。

(5) 用户管理命令，例如 adduser、userdel 等。

(6) 文档操作命令，例如 csplit、sort 等。

(7) 网络通信命令，例如 netstat、ifconfig 等。

(8) 程序开发命令，例如 cc、link 等。

(9) X Window 管理命令，例如 startx、XE86Setup 等。

用户通常会使用到的命令大多放在/usr/sbin、/usr/bin、/sbin、/bin 目录下，使用系统提供的 man 命令可以格式化并显示某一命令的联机帮助。

Linux 交互式使用命令或允许用户自己编写 Shell 程序以采用批处理方式操作。

Linux Shell 是一种交互型命令解释程序，也是一种命令级程序设计语言解释系统，它允许用户编制带形式参数的批命令文件，称作 Shell 脚本或 Shell 程序。在 Linux 下，可以像执行任何其他命令一样直接输入其名称来执行一个 Shell 程序。一个 Shell 程序由以下部分组成：

- 命令或其他 Shell 程序；
- 位置参数；
- 变量及特殊字符；
- 表达式比较；
- 控制流语句，例如 while、case 等；
- 函数。

例如，把目录中每个文件在一个子目录中建立备份的 Shell 程序如下：

```
mkdir backup
```

```
for file in 'ls'
do
    cp $file backup/$file
    if [$?-ne 0]; then
        echo "copying $file error"
    fi
done
```

在这个例子中，首先在当前目录下创建了一个子目录 backup，然后在其中循环地建立当前目录下所有文件的备份。其中 mkdir、cp、echo 等是 Linux 命令，for 是 Shell 的循环语句，$ file 是自定义变量，$？ 是 Shell 内部变量，-ne 是 Shell 的表达式比较操作符。

Linux Shell 可定制性强，支持命令广，具有良好的作业控制能力，编写的 Shell 命令又可通过脚本的形式重新组合使用，完成对用户的计算环境定制等，功能十分方便。但 Shell 脚本作为一种解释程序，执行效率低，操作粒度粗，不适合直接操作计算机的存储和 I/O 等设备。

Linux 窗口系统的命令操作和控制非常简单，它们是一个以图表等显示的命令集合，用户可以用鼠标和键盘方便地和系统进行交互。

### 2.4.2 Windows 的命令控制界面

Windows 视窗系统是在 DOS、Windows 3.2 等基础上不断演化、发展而成的。网页 http://www.microsoft.com/windows/WinHistoryIntro.mspx 上详细地介绍了 Windows 用户界面及其发展历史。

Windows 的命令控制界面可以分为两大部分，即命令解释器部分（相当于 Linux 的 Shell）和窗口交互部分。

Windows 窗口部分主要是利用鼠标或键盘，通过直观的方式对图形化界面进行操作，这里不再详细介绍。

Windows 通过自带的命令行解释器 cmd.exe 为用户提供了功能强大的命令行控制界面，用户可以通过输入命令来对 Windows 操作系统进行控制和使用。这些命令包括了从 MS-DOS 保留下来的一些基本命令以及 Windows 自有的操作命令，主要分为以下几类：

（1）系统信息命令，例如 Time、Date、Mem、Driverquery 和 SystemInfo 等。

（2）系统操作命令，例如 Shutdown、Runas 和 Taskkill 等。

（3）文件系统命令，例如 Copy、Del 和 Mkdir 等。

（4）网络通信命令，例如 Ping、Netstat 和 Route 等。

每个命令有不同的功能，不同的命令可以按照下面的形式进行组合从而形成新的命令：

Command1 & Command2：用来分隔一个命令行中的多个命令。即 Cmd.exe 运行第一个命令，然后运行第二个命令。

Command1 && Command2：只有在符号 && 前面的命令运行成功时，才运行符号 && 后面的命令。

Command1 || Command2：只有在符号||前面的命令运行失败时，才运行符号||后面的命令。

(Command1 & Command2)：用户分组或嵌套多个命令。

Command1 parameter1;parameter2：用分号(;)分隔命令参数。

用上面的命令或者命令组合，用户就可以完成需要的功能。在使用时，通常有下面两种方式。

(1) 直接在命令行输入命令

用户运行 cmd. exe，进入命令行界面，在命令行提示符下输入命令。例如：

```
Systeminfo&mem
```

显示当前系统的属性和配置等，然后显示当前内存的使用情况。

(2) 使用批处理文件

批处理文件是无格式的文本文件，它包含一条或多条命令，其文件扩展名为. bat 或. cmd。在命令提示符下输入批处理文件的名称，cmd. exe 就会按照该文件中各个命令出现的顺序来逐个运行它们。例如，用户在当前目录下新建一个批处理文件 exam1. bat，其内容如下：

```
@echo off
mkdir test
echo hello
pause
```

其中，@表示不显示当前命令本身，echo off 表示后面的命令都不显示，mkdir test 表示在当前目录下新建一个 test 文件夹，echo hello 表示打印 hello 字符，pause 表示暂停执行，等待用户响应。

用户可以直接用鼠标双击 exam1. bat 的图标执行批处理，也可以先运行 cmd. exe，在命令提示符下输入 exam1 来执行批处理，脚本中的命令将顺序执行。

批处理文件之间是可以相互调用和传递参数的，这样，用户就可以将单元功能模块连接起完成更为复杂的功能，避免了用户手工进行功能模块命令的输入。例如，在当前路径下还有两个批处理文件。

exam2. bat 的内容如下：

```
@echo off
mem> %1/meminfo.txt
echo generate memory info ok!
```

exam3. bat 的内容如下：

```
@echo off
type %1\* .txt
echo type ok!
```

其中，%1 表示第一个输入参数，exam2. bat 的功能是将系统当前的内存使用情况保存到以输入参数 1 为名字的文件夹下的 meminfo. txt 文件中，如果 meminfo. txt 不存在，则进行创建；exam3. bat 的功能是将以输入参数 1 为名字的文件夹下的所有 txt 文件的内容打印出来。

将前面的 exam1. bat 进行改写：

```
@echo off
mkdir test
call exam2.bat test
call exam3.bat test
echo call ok!
pause
```

这样，在 exam1. bat 中对 exam2 和 exam3 进行了调用，使用 test 文件夹下的 meminfo . txt，其中保存了系统当前的内存使用情况，在屏幕上将其内容打印出来。

用鼠标双击 exam1. bat 的图标或者在命令行模式下运行 exam1. bat，运行结果如图 2. 6 所示。运行结束后，在当前文件夹下新建了 test 文件夹，其下有 meminfo. txt 文件，存放的文本是系统当前的内存使用信息。

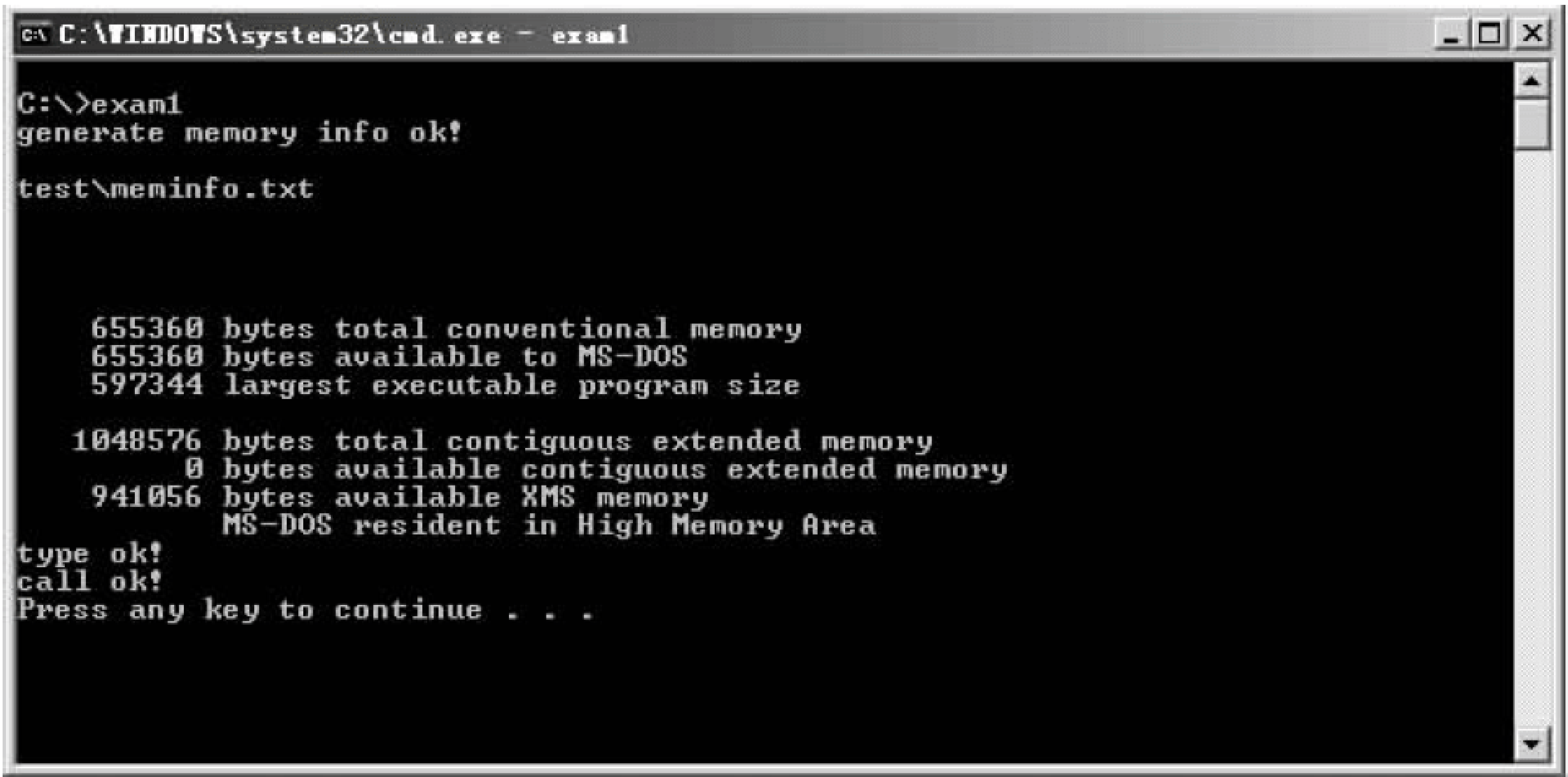



图 2. 6　相互调用批处理示例

## 2. 5　系 统 调 用

系统调用是操作系统提供给编程人员的唯一接口。编程人员利用系统调用，在源程序一级动态请求和释放系统资源，调用系统中已有的系统功能来完成那些与计算机硬件部分相关的工作以及控制程序的执行速度等。因此，系统调用像一个黑箱子那样，对用户屏蔽了操作系统的具体动作而只提供有关的功能。事实上，命令控制界面也是在系统调用的基础上开发而成的。

系统调用大致可分为如下几类：

(1) 设备管理。用来请求和释放有关设备以及启动设备操作等。

(2) 文件管理。包括对文件的读、写、创建和删除等。

(3) 进程控制。进程是一个在功能上独立的程序的一次执行过程。进程控制的有关系统调用包括进程创建、进程执行、进程撤销、执行等待和执行优先级控制等。

(4) 进程通信。该类系统调用用于在进程之间传递消息或信号。

(5) 存储管理。包括调查作业占据内存区的大小、获取作业占据内存区的始址等。

(6) 线程管理。包括线程的创建、调度、执行和撤销等。

不同的系统提供不同的系统调用。一般，每个系统为用户提供几十到几百条系统调用。

为了提供系统调用功能，操作系统内必须有事先编制好的实现这些功能的子程序或过程。同时，为了保证操作系统程序不被用户程序破坏，一般操作系统都不允许用户程序直接访问操作系统的系统程序和数据。编程人员如何调用操作系统内部的程序或数据？这需要有一个类似于硬件中断的处理机制。当用户使用系统调用时，产生一条相应的指令，处理机在执行到该指令时发生相应的中断，并发出有关信号给该处理机制。该处理机制在收到了处理机发来的信号后，启动相关的处理程序去完成该系统调用所要求的功能。

在系统中为控制系统调用服务的处理机构称为陷阱(trap)处理机构。与此相对应，把由于系统调用引起处理机中断的指令称为陷阱指令(或称访管指令)。在操作系统中，每个系统调用都对应一个事先给定的功能号，例如0、1、2和3等。在陷阱指令中必须包括对应系统调用的功能号。而且，在有些陷阱指令中，还带有传递给陷阱处理机构和内部处理程序的有关参数。

为了实现系统调用，系统设计人员还必须为实现各种系统调用功能的子程序编制入口地址表，每个入口地址都与相应的系统子程序名对应起来。然后，由陷阱处理程序把陷阱指令中所包含的功能号与该入口地址表中的有关项对应起来，从而由系统调用功能号驱动有关系统子程序执行。

由于在系统调用处理结束之后，用户程序还需利用系统调用的返回结果继续执行，因此，在进入系统调用处理之前，陷阱处理机构还需保存处理机现场。再者，在系统调用处理结束之后，陷阱处理机构还要恢复处理机现场。在操作系统中，处理机的现场一般被保护在特定的内存区或寄存器中。系统调用的处理过程如图2.7所示。

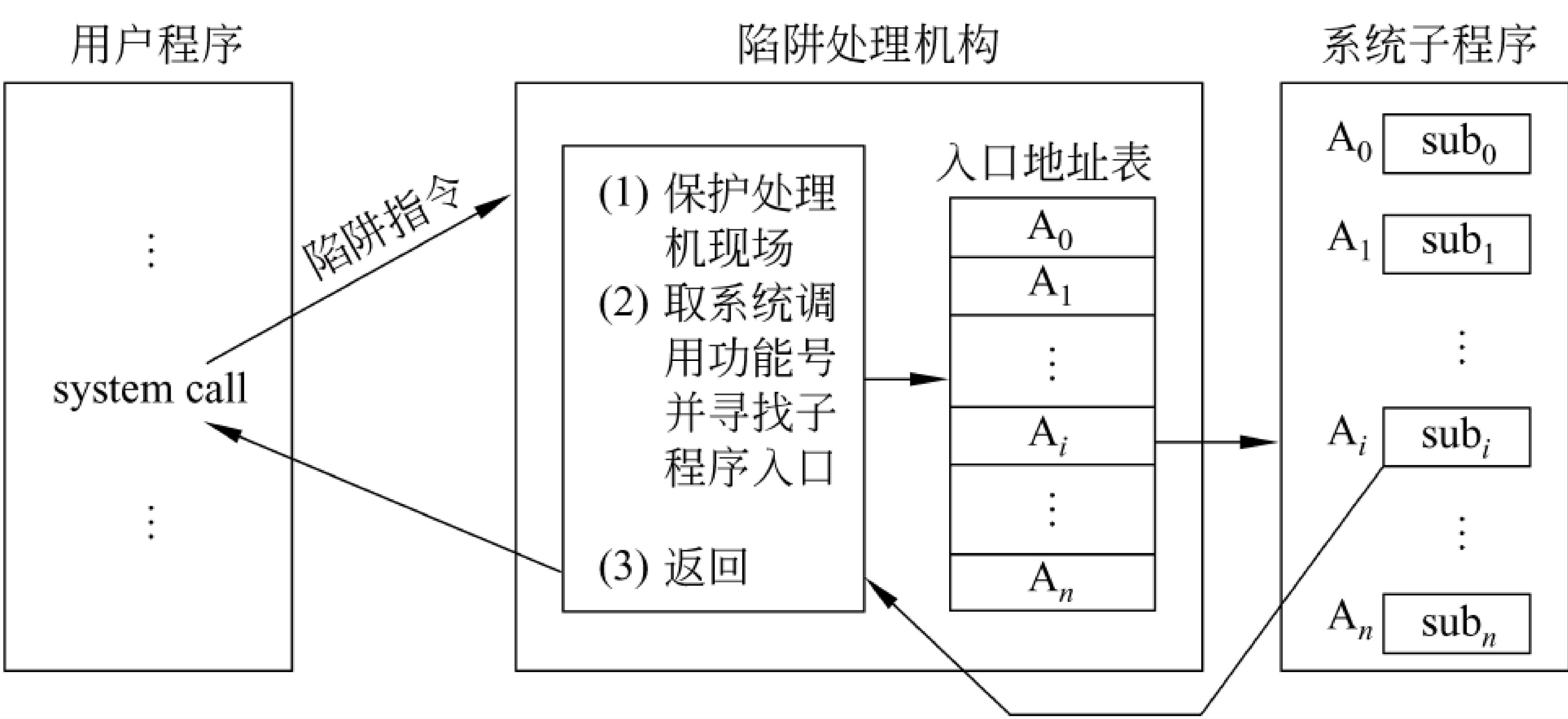



图2.7　系统调用的处理过程

有关系统调用的另一个问题是参数传递问题。不同的系统调用需要传递给系统子程序以不同的参数。而且，系统调用的执行结果也要以参数形式返回给用户程序。那么，怎样实现用户程序和系统程序之间的参数传递呢？下面介绍几种常用的实现方法。一种是由陷阱指令自带参数。一般来说，一条陷阱指令的长度总是有限的，而且，该指令还要携带一个系统调用的功能号，从而，陷阱指令只能自带极有限的几个参数进入系统内部。另一种办法是通过使用有关通用寄存器来传递参数。显然，这些寄存器应是系统程序和用户程序都能访问的。不过，由于寄存器长度也是较短的，从而无法传递较多的参数。因此，在系统调用较多的系统中，大多在内存中开辟专用堆栈区来传递参数。

另外,在系统发生访管中断或陷阱中断时,为了不让用户程序不直接访问系统程序,反映处理机硬件状态的处理机状态字(PSW)中的相应位要从用户执行模式转换为系统执行模式。这一转换在发生访管中断时由硬件自动实现。一般把处理机在用户程序中执行称为用户态,而把处理机在系统程序中执行称为系统态。

## 2.6 Linux 和 Windows 的系统调用

### 2.6.1 Linux 系统调用

Linux 提供了丰富的系统调用。每个系统调用由两部分组成:核心函数部分提供实现系统调用功能的共享代码,作为操作系统的核心程序驻留在内存中;接口函数部分提供给应用程序 API 接口,它把系统调用号和入口参数地址传送给相应的核心函数。

Linux 提供多达上百种系统调用,从功能上大致可分为如下几类:

- 设备管理的系统调用。申请设备,释放设备,设备 I/O 和重定向,设备属性获取及设置,逻辑上连接和释放设备。
- 文件系统操作的系统调用。建立文件,删除文件,打开文件,关闭文件,读写文件,获得和设置文件属性。
- 进程控制的系统调用。终止或异常终止进程,载入和执行进程,创建和撤销进程,获取和设置进程属性。
- 存储管理的系统调用。申请内存和释放内存。
- 管理用的系统调用。获取和设置日期及时间,获取和设置系统数据。
- 通信的系统调用。建立和断开通信连接,发送和接收消息,传送状态信息,连接和断开远程设备。

编程人员可以使用不同的系统调用来实现自己所需要的功能。例如,下面是一个使用系统调用打开(open)、读(read)、写(write)、关闭(close)等完成文件复制的例子:

```
#include <fcnt1.h>
#include <sys/stat.h>
#define SIZE 1
void filecopy(char * Infile,char * Outfile)
{
     char Buffer[SIZE];
    int In_fh,Out_fh,Count;
    if((In_fh=open(Infile,O_RDONLY))==-1)   /* 以只读模式打开输入文件 */
          printf("Opening Infile");
    if(Out_fh=open(Outfile,(O_WRONLY|O_CREAT|O_TRUNC),(S_IRUSR|S_IWUSR))==-1)
                                            /* 以读写模式新建一个文件 */
          printf("Opening Outfile");
    while((Count=read(In_fh,Buffer,sizeof(Buffer)))>0)
                                            /* 循环地向缓冲区读入输入文件内容 */
    if(write(Out_fh,Buffer,Count)!=Count)   /* 将缓冲区读入的内容写到输出文件中 */
          printf("Writing data");
```

```
    if(Count==-1)
        printf("Reading data");
    close(In_fh);                           /* 关闭输入文件 */
    close(Out_fh);                          /* 关闭输出文件 */
}
```

读者可以使用 Linux 系统调用编写自己的应用程序。

### 2.6.2 Windows 系统调用

Windows 操作系统也提供了丰富的系统调用。这些系统调用被进一步编写成不同的库函数后放入动态链接库(DLL)中。这些库函数构成了 Windows 操作系统提供给程序员的编程接口。这个编程接口被称为应用编程接口,(Application Programming Interface, API)。

从 Windows 1.0 以来,系统就提供了 API 函数的调用。随着系统的不断升级,API 函数也不断地得到扩充,从 Win 16 API 发展到 Win 32 API,高版本系统对低版本系统的 API 的函数都提供了支持。在 Windows 2003 中提供的 API 函数已经扩充到了几千个。

常用的 API 函数调用分为如下几类:

(1) 窗口管理类。向应用程序提供了一些创建和管理用户界面的方法,可以使用窗口管理函数创建和使用窗口来显示输出、提示用户进行输入以及完成其他一些与用户进行交互所需的工作,大多数应用程序都至少要创建一个窗口。主要包括按钮函数(Button)、光标函数(Cursor)和对话框函数(Dialog Box)等的调用。

(2) 图形设备接口(GDI)类。提供了一系列的函数和相关的结构,应用程序可以使用它们在显示器、打印机或其他设备上生成图形化的输出结果。使用 GDI 函数可以绘制直线、曲线、闭合图形、路径、文本以及位图图像。主要包括位图函数(Bitmap)、笔刷函数(Brush)和颜色函数(Color)等的调用。

(3) 系统服务类。为应用程序提供了访问计算机资源以及底层操作系统特性的手段,比如访问内存、文件系统、设备、进程和线程。应用程序使用系统服务函数来管理和监视它所需要的资源。主要包括内存管理函数(Memory Management)、文件函数(File)以及进程和线程函数(Process and Thread)等的调用。

(4) 国际特性类。帮助用户编写国际化的应用程序,提供给用户将应用程序本地化的一些功能。主要包括 Unicode 和字符集函数(Unicode and Character Set)以及输入方法编辑器函数(Input Method Editor)等的调用。

(5) 网络服务类。允许网络上的不同计算机的应用程序之间进行通信,帮助在网络中的各计算机上创建和管理共享资源的连接,例如共享目录和网络打印机。主要包括 Windows 网络函数、Windows 套接字(Socket)、NetBIOS、RAS、SNMP 和 Net 函数等的调用。

例如:

GDI32.DLL 给出了屏幕显示及打印功能的函数集;

USER32.DLL 给出了鼠标、键盘、通信端口、声音、时钟功能的函数集;

KERNEL32.DLL 给出了文件及内存管理(核心部分)功能的函数集;

MPR.DLL 给出了 Windows 3.2 网络接口库。

下面是一个使用 Windows API 编程的例子。在该例子中，先创建一个文件，然后向创建的文件中写入字符串，再从该文件中将字符串读取出来并通过 MessageBox 显示。

```
#include<windows.h>
//入口函数
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine,
int iCmdshow)
{
    HANDLE hFile;
    LPTSTR lpBuffer="Hello World !";
    //创建文件
    hFile=CreateFile("C:\\File.txt",GENERIC_READ | GENERIC_WRITE,
                     0,NULL,OPEN_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
    CloseHandle(hFile);
    TCHAR szBuf[128];
    DWORD dwRead;
    DWORD dwWritten;
    //打开文件
    hFile=CreateFile("C:\\File.txt",GENERIC_READ | GENERIC_WRITE,
                   0,NULL,OPEN_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
    //向文件中写入一个字符串
    WriteFile(hFile,lpBuffer,strlen(lpBuffer)+1,&dwRead,NULL);
    SetFilePointer(hFile,0,NULL,FILE_BEGIN);
    //从文件中读出一个字符串并将它显示在对话框中
    if(ReadFile(hFile,szBuf,strlen(lpBuffer)+1,&dwWritten,NULL));
    {
        messagebox(NULL,szBuf,"EXAM",MB_OK);
    }
    CloseHandle(hFile);
}
```

在上述程序代码中，使用了 Windows API 的系统服务类中有关文件的操作函数，如 CreateFile、WriteFile 及 ReadFile 等完成相关的文件操作。

读者可从网站 www.msdn.com 得到更多的关于 Windows API 的知识。

## 本章小结

本章简要介绍了操作系统的用户界面。操作系统的用户界面是评价一个操作系统优劣的重要指标。操作系统的用户界面包括命令控制界面和编程界面两部分，其中命令控制界面是基于编程界面，也就是系统调用之上开发而成的。

操作系统的命令控制界面已从早期的脱机控制方式（批处理系统）和联机控制方式（分时系统）转向多窗口、菜单、按钮以及声控等图形化多媒体方式。命令控制界面的革命与进步是操作系统最显著的变化之一。

系统调用是操作系统提供给编程人员的唯一接口。编程人员通过系统调用使用操作系统内核所提供的各种功能。系统调用的执行不同于一般用户程序的执行。系统调用执行是在核心态下执行系统子程序，而用户程序则是在用户态下执行。一般来说，操作系统提供的系统调用越多，功能也就越丰富，系统也就越复杂。

本章还简要介绍了 Linux 和 Windows 系统的用户界面。

## 习　　题

2.1　什么是作业和作业步？

2.2　作业由哪几部分组成？这几部分各有什么功能？

2.3　作业的输入方式有哪几种？各有何特点？

2.4　试述 spooling 系统的工作原理。

2.5　操作系统为用户提供哪些接口？它们的区别是什么？

2.6　作业控制方式有哪几种？调查你周围的计算机的作业控制方式。

2.7　什么是系统调用？系统调用与一般用户程序有什么区别？与库函数和实用程序又有什么区别？

2.8　简述系统调用的实现过程。

2.9　为什么说分时系统没有作业的概念？

2.10　Linux 操作系统为用户提供哪些接口？试举例说明。

2.11　在你周围装有 Linux 系统的计算机上，查看有关 Shell 的基本命令，并编写一个简单的 Shell 程序，完成一个已有数据文件的复制和打印。

2.12　用 Linux 文件读写的相关系统调用编写 copy 程序。

2.13　用 Windows 的 DLL 接口编写 copy 程序。

# 第 3 章　进 程 管 理

## 3.1　进程的概念

现代操作系统的重要特点是在保证安全的前提下,程序并发执行,以及系统所拥有的资源被共享和系统的用户随机地使用。这 3 个特点是互相联系和互相依赖的,它们是互相独立的用户如何使用有限的计算机系统资源的反映。通常,操作系统的重要任务之一是使用户充分、有效地利用系统资源。采用一个什么样的概念来描述计算机程序的执行过程和作为资源分配的基本单位,才能充分反映操作系统的执行并发、资源共享及用户随机的特点呢? 这个概念就是进程。为了讲清进程的概念以及引入进程概念的必要性等,下面将从操作系统的特点讲起。

### 3.1.1　程序的并发执行

**1. 程序**

程序(program)描述计算机所要完成的具有独立功能的,并在时间上按严格次序前后相继的计算机操作序列集合,是一个静态的概念。它体现了编程人员要求计算机完成相应功能时所应该采取的顺序步骤。

**2. 程序的顺序执行**

程序只有经过执行才能得到结果。程序的执行又可以分为顺序执行和并发执行。计算机的 CPU 是通过时序脉冲来控制顺序执行指令的。其执行过程可以描述为

```
repeat IR←M[PC]
    PC←PC+1
    <Execute (instruction in IR)>
until CPU halt
```

这里 IR 为指令寄存器,PC 为程序计数器,M 为存储器。显然,程序的顺序性与计算机硬件的顺序性是一致的。我们把一个具有独立功能的程序独占处理机直至最终结束的过程称为程序的顺序执行。程序的顺序执行具有如下特点:

1) 顺序性

程序顺序执行时,其执行过程可看作一系列严格按程序规定的状态转移过程,也就是每执行一条指令,系统将从上一个执行状态转移到下一个执行状态,且上一条指令的执行结束是下一条指令执行开始的充分必要条件。

2) 封闭性

程序执行得到的最终结果由给定的初始条件决定,不受外界因素的影响。

3) 可再现性

顺序执行的最终结果可再现是指它与执行速度无关。只要输入的初始条件相同,则无

论何时重复执行该程序都会得到相同的结果。

**3. 多道程序系统中程序执行环境的变化**

如果每台计算机在任一时刻只处理一个具有独立功能的程序，操作系统的设计和功能都将变得非常简单，因为在这样的系统中不存在资源共享、程序的并发执行以及用户执行的随机性问题。但是，在许多情况下，计算机需要能够同时处理多个具有独立功能的程序。批处理系统、分时系统、实时系统以及网络与分布式系统等都是这样的系统。因此，每个程序在执行时都应考虑其执行环境的变化。这样的执行环境具有下述 3 个特点。

1）独立性

在多道环境下执行的每道程序都是逻辑上独立的，它们之间不存在逻辑上的制约关系。也就是说，如果有充分的资源保证，则每道程序都可以独立执行，且执行速度与其他程序无关，执行的起讫时间也是独立的。

2）随机性

在多道程序环境下，特别是在多用户环境下，程序和数据的输入与执行开始时间都是随机的。在实时系统中更是如此，它被要求在一个被允许的短时间内对随机的输入做出反应。输入与程序执行开始时间的随机性形成了操作系统必须同时处理多道程序的客观要求。

3）资源共享性

这里所说的资源，既包括硬件资源，也包括软件资源。硬件资源包括 CPU、输入输出设备和存储器等。软件资源除了指各种例程之外，还包括各种可共享的数据。显然，任何一个计算机系统中的软、硬件资源都是有限的，特别是在单 CPU 系统中。一般来说，多道环境下执行程序的道数总是要超过计算机系统中 CPU 的个数，单 CPU 系统更是如此。显然，受 CPU 个数的限制，由随机性引起的需同时执行的 $N$ 道（$N \geq 1$）程序只能共享系统中已有的 CPU。在单 CPU 系统中，则有 $N-1$ 道程序处于等待 CPU 的状态。同理，输入输出设备有限将导致这些设备被共享，内存有限将导致内存被共享等。资源共享将导致对进程执行速度的制约。

**4. 程序的并发（concurrent）执行**

1）什么是程序的并发执行

所谓并发执行，是为了增强计算机系统的处理能力和提高资源利用率所采取的一种同时操作技术。程序的并发执行可进一步分为两种：第一种是多道程序系统的程序执行环境变化所引起的多道程序的并发执行。如前所述，在多道程序系统环境下，各道程序在逻辑上独立，具备了独立执行的条件。而程序与数据输入的随机性以及执行起始时间的随机性又导致了多道程序要求同时执行的客观要求。由于资源的有限性，多道程序的并发执行总是伴随着资源的共享与竞争。从而制约各道程序的执行速度。而无法做到在微观上，也就是在指令级上的同时执行。因此，尽管多道程序的并发执行在宏观上是同时进行的，但在微观上仍是顺序执行的；第二种并发执行是在某道程序的几个程序段中（例如几个程序）包含着一部分可以同时执行或顺序颠倒执行的代码。例如，语句

```
read(a);
read(b);
```

既可以同时执行，也可颠倒次序执行。也就是说，对于这样的语句，同时执行不会改变顺序

程序所具有的逻辑性质。因此,可以采用并发执行来充分利用系统资源以提高计算机的处理能力。

综上所述,程序的并发执行可总结为:一组在逻辑上互相独立的程序或程序段在执行过程中,其执行时间在客观上互相重叠,即一个程序段的执行尚未结束,另一个程序段的执行已经开始的这种执行方式。

程序的并发执行不同于程序的并行执行。程序的并行执行是指一组程序按独立的、异步的速度执行。并行执行不等于时间上的重叠。可以将并发执行过程描述为

```
S0
Cobegin
P1;P2;…;Pn
Coend
Sn
```

这里,S0,Sn 分别表示并发程序段 P1,P2,…,Pn 开始执行前和并发执行结束后的语句。即:先执行 S0,再并发执行 P1,P2,…,Pn;当 P1,P2,…,Pn 全部执行完毕后,再执行 Sn。P1,P2,…,Pn 也可以由同一程序段中的不同语句组成。

1966 年 Bernstein 提出了两个相邻语句 $S_1$、$S_2$ 可以并发执行的条件:

将程序中任一语句 $S_i$ 划分为两个变量的集合 $R(S_i)$和 $W(S_i)$。其中

$$R(S_i)=\{a_1\ a_2\cdots\ a_m\}$$

$a_j(j=1,2,\cdots,m)$是语句 $S_i$ 在执行期间必须对其进行读写的变量;

$$W(S_i)=\{b_1\ b_2\cdots\ b_n\}$$

$b_j(j=1,2,\cdots,n)$是语句 $S_i$ 在执行期间必须对其进行修改、访问的变量。

如果对于语句 $S_1$ 和 $S_2$,有

① $R(S_1)\cap W(S_2)=\{\phi\}$

② $W(S_1)\cap R(S_2)=\{\phi\}$

③ $W(S_1)\cap W(S_2)=\{\phi\}$

同时成立,则语句 $S_1$ 和 $S_2$ 是可以并发执行的。

2）程序的并发执行所带来的影响

程序的并发执行充分地利用了系统资源,从而提高了系统的处理能力,这是并发执行好的一方面。但是,正如前面所提到的那样,由于系统资源有限,程序的并发执行必然导致资源共享和资源竞争,从而改变程序的执行速度。那么,由资源共享和竞争所引起的程序执行速度的改变是否会对程序的最终结果带来不利的影响呢? 也就是说,是否会保持用户所期望的执行结果的封闭性和可再现性呢? 如果并发执行的各程序段中语句或指令满足上述 Bernstein 的 3 个条件,则认为并发执行不会对执行结果的封闭性和可再现性产生影响。但在一般情况下,系统要判定并发执行的各程序段是否满足 Bernstein 条件是相当困难的。如果并发执行的程序段不按照特定的规则和方法进行资源共享和竞争,则其执行结果将不可避免地失去封闭性和可再现性。下面的例子说明了这一点。

**例**:设有堆栈 S,栈指针 top,栈中存放内存中相应的数据块地址(如图 3.1(a)所示)。设有两个程序段 getaddr(top)和 reladdr(blk),其中 getaddr(top)从给定的 top 所指的栈中取出相应的内存数据块地址,而 reladdr(blk)则将内存数据块地址 blk 放入堆栈 S 中。

getaddr(top)和 reladdr(blk)可分别描述为

```
procedure getaddr(top)
    begin
    local r
    r ← (top)
    top ← top-1
    return(r)
    end
procedure reladdr(blk)
    begin
    top ← top+1
    (top)← blk
    end
```

显然,如果上例中的 getaddr 和 reladdr 程序段采用顺序执行,其执行结果具有封闭性和可再现性;但如果对这两个程序段采用并发执行,则在单 CPU 系统中,将有可能出现下述情况:

首先,程序段 reladdr 开始执行,准备释放内存数据块地址入栈。然而,当 reladdr 执行到 top←top+1 语句时(如图 3.1(b)所示),程序段 getaddr 也开始执行且抢占了处理机,从而程序段 reladdr 停在 top←top+1 处等待处理机。getaddr 程序段的执行目的是要从对应的堆栈指针 top 所指的栈格中取出一个内存数据块地址,显然,由于 reladdr 程序段的执行将指针 top 升高了一格且未放进适当的数据,getaddr 的执行结果是失败的(如图 3.1(c)所示)。另外,如果改变程序段 getaddr 和 reladdr 的执行顺序或执行速度,又可得到不同的执行结果。这说明了如下问题:在某些情况下,程序的并发执行使得其执行结果不再具有封闭性和可再现性,且可能造成程序出现错误。

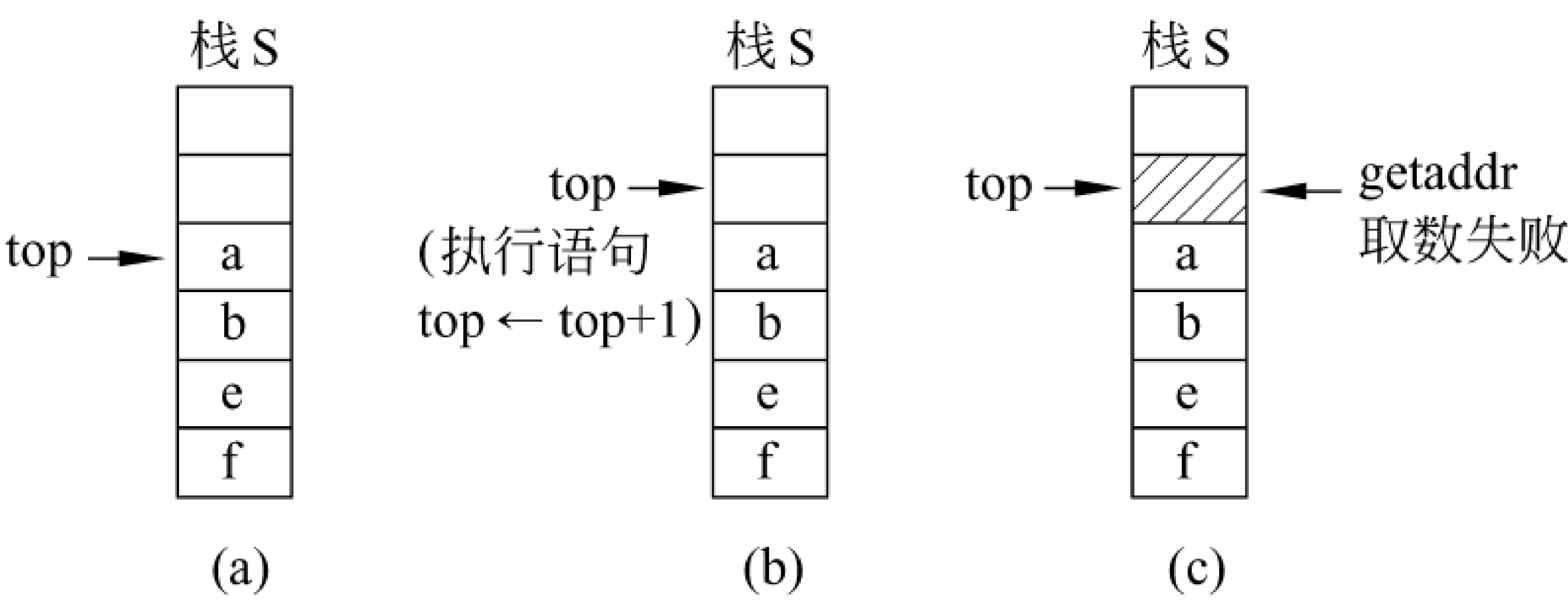



图 3.1　堆栈的取数和存数过程

上例中的程序段并发执行出现错误结果,是由于两程序段共享资源堆栈 S,从而使得执行结果受执行速度影响。一般情况下,并发执行的各程序段如果共享软、硬件资源,都会造成其执行结果受执行速度影响的局面。显然,这是程序设计人员不希望看到的。为了使得在并发执行时不出现错误结果,必须采取某些措施来制约、控制各并发程序段的执行速度。这在操作系统程序设计中尤其重要,因为操作系统用户随机性与各道程序逻辑独立的特点将使得每个用户程序所使用的软、硬件资源都受到其他并发程序的共享和竞争,从而得到非预料的或不正确的结果。为了控制和协调各程序段执行过程中的软、硬件资源的共享和竞争,显然,必须应该有一个描述各程序段执行过程和共享资源的基本单位。从上述讨论可以

看出，由于程序的顺序性、静态性以及孤立性，用程序段作为描述其执行过程和共享资源的基本单位既增加操作系统设计和实现的复杂性，也无法反映操作系统所应该具有的程序段执行的并发性、用户随机性以及资源共享等特征。也就是说，用程序作为描述其执行过程以及共享资源的基本单位是不合适的。需要有一个能描述程序的执行过程且能用来共享资源的基本单位，这个基本单位被称为进程（或任务）。

### 3.1.2 进程的定义

进程的概念是 20 世纪 60 年代初期首先在 MIT 的 Multics 系统和 IBM 公司的 TSS/360 系统中引用的。从那以后，人们对进程下过许多各式各样的定义。现列举其中几种：

（1）进程是可以并发执行的计算部分（S. E. Madnick，J. T. Donovan）。

（2）进程是一个独立的可以调度的活动（E. Cohen，D. Jofferson）。

（3）进程是一个抽象实体，当它执行某个任务时，将要分配和释放各种资源（P. Denning）。

（4）行为的规则称为程序，程序在处理机上执行时的活动称为进程（E. W. Dijkstra）。

（5）一个进程是一系列逐一执行的操作，而操作的确切含义则有赖于以何种详尽程度来描述进程（Brinch Hansen）。

以上进程的定义，尽管各有侧重，但在本质上是相同的，即主要注重进程是一个动态的执行过程这一概念。也可以这样定义进程：并发执行的程序在执行过程中分配和管理资源的基本单位。

进程和程序是两个既有联系又有区别的概念，它们的区别和联系可简述如下：

（1）进程是一个动态概念，而程序则是一个静态概念。程序是指令的有序集合，没有任何执行的含义。而进程则强调执行过程，它动态地被创建，并被调度执行后消亡。如果把程序比作菜谱，那么进程则是按照菜谱炒菜的过程。

（2）进程具有并发特征，而程序没有。由进程的定义可知，进程具有并发特征的两个方面，即独立性和异步性。也就是说，在不考虑资源共享的情况下，各进程的执行是独立的，执行速度是异步的。显然，由于程序不反映执行过程，所以不具有并发特征。

（3）进程是竞争计算机系统资源的基本单位，从而其并发性受到系统自己的制约。这里，制约就是对进程独立性和异步性的限制。

（4）不同的进程可以包含同一程序，只要该程序所对应的数据集不同。

## 3.2 进程的描述

一个进程是一个程序对某个数据集的执行过程，是分配资源的基本单位。那么，从处理机的活动角度来看，又如何识别描述程序执行活动的进程呢？显然，系统中需要有描述进程存在和能够反映其变化的物理实体，即进程的静态描述。进程的静态描述由 3 部分组成：进程控制块（PCB），有关程序段和该程序段对其进行操作的数据结构集。进程控制块包含了有关进程的描述信息、控制信息以及资源信息，是进程动态特征的集中反映。系统根据 PCB 感知进程的存在，通过 PCB 中所包含的各项变量的变化掌握进程所处的状态，以达到控制进程活动的目的。由于进程的 PCB 是系统感知进程的唯一实体，因此，在几乎所有的

多道操作系统中，一个进程的 PCB 结构都是全部或部分常驻内存的。

进程的程序部分描述进程所要完成的功能。而数据结构集是程序在执行时必不可少的工作区和操作对象。这两部分是进程完成所需功能的物质基础。由于进程的这两部分内容与控制进程的执行及完成进程功能直接有关，因而，在大部分多道操作系统中，这两部分内容放在外存中，直到该进程执行时再调入内存。下面分别介绍进程的 PCB 结构、程序与数据结构集。

### 3.2.1 进程控制块

如上所述，进程控制块(PCB)包含一个进程的描述信息、控制信息及资源信息，有些系统中还有进程调度等待所使用的现场保护区。PCB 集中反映一个进程的动态特征。在进程并发执行时，由于资源共享，带来各进程之间的相互制约。显然，为了反映这些制约关系和资源共享关系，在创建一个进程时，应首先创建其 PCB，然后才能根据 PCB 中的信息对进程实施有效的管理和控制。当一个进程完成其功能之后，系统则释放 PCB，进程也随之消亡。

一般来说，根据操作系统的要求不同，进程的 PCB 所包含的内容会多少有所不同。但是，下面所示的基本内容是必需的。

**1. 描述信息**

描述信息主要包括下列几项。

(1) 进程名或进程标识号

每个进程都有唯一的进程名或进程标识号。在识别一个进程时，进程名或进程标识号代表该进程。

(2) 用户名或用户标识号

每个进程都隶属于某个用户，用户名或用户标识号有利于资源共享与保护。

(3) 家族关系

在有的系统中，进程之间互成家族关系。因此，PCB 中相应的项描述其家族关系。

**2. 控制信息**

控制信息包括下列几项。

1) 进程当前状态

进程当前状态说明进程当前处于何种状态。进程在活动期间可分为初始态、就绪态、执行态、等待状态和终止状态。任一进程在任一时刻只能处于这 5 种状态中的一种。这 5 种状态的含义是：执行态表示该进程占有处理机，就绪态表示该进程准备占有处理机，而等待状态则表示进程因某种原因而暂时不能占有处理机。在有的系统中，等待状态会进一步划分为不同原因或不同地点(内存或外存)的等待状态。有关进程的状态在 3.3 节中进一步讨论。

2) 进程优先级

进程优先级是选取进程占有处理机的重要依据。与进程优先级有关的 PCB 表项有以下几项。

(1) 占有 CPU 时间；

(2) 进程优先级偏移；

(3) 占据内存时间，等等。

3) 程序开始地址

程序开始地址规定该进程的程序以此地址开始执行。

4) 各种计时信息

给出进程占有和利用资源的有关情况。

5) 通信信息

通信信息用来说明该进程在执行过程中与别的进程所发生的信息交换情况。

**3. 资源管理信息**

PCB中包含最多的是资源管理信息，包括有关存储器的信息、使用输入输出设备的信息和有关文件系统的信息等。这些信息的具体内容如下：

(1) 占用内存大小及其管理用数据结构指针，例如内存管理中所用到的进程页表指针等。

(2) 在某些复杂系统中，还有对换或覆盖用的有关信息，如对换程序段长度和对换外存地址等。这些信息在进程申请、释放内存中使用。

(3) 共享程序段大小及起始地址。

(4) 输入输出设备的设备号，所要传送的数据长度、缓冲区地址、缓冲区长度及所用设备的有关数据结构指针等。这些信息在进程申请释放设备进行数据传输时使用。

(5) 指向文件系统的指针及有关标识等。进程可使用这些信息对文件系统进行操作。

**4. CPU 现场保护结构**

当前进程因等待某个事件而进入等待状态或因某种事件发生被中止在处理机上的执行时，为了以后该进程能在被打断处恢复执行，需要保护当前进程的 CPU 现场。PCB 中设有专门的 CPU 现场保护结构，以存储退出执行时的进程现场数据。

总之，PCB 是系统感知进程存在的唯一实体。通过对 PCB 的操作，系统为有关进程分配资源从而使得有关进程得以被调度执行；而完成进程所要求功能的程序段的有关地址，以及程序段在进程过程中因某种原因被停止执行后的现场信息也都在 PCB 中。最后，当进程执行结束后，则通过释放 PCB 来释放进程所占有的各种资源。

由于 PCB 中包含较多的信息，因此，一个 PCB 表往往要占据较大的存储空间(一般占几百到几千个字节)。在有的系统中，为了减少 PCB 对内存的占用量，只允许 PCB 中最常用的部分，如 CPU 现场保护、进程描述信息和控制信息等常驻内存。PCB 结构中的其他部分则存放于外存之中，待该进程将要执行时与其他数据一起装入内存。

### 3.2.2 进程上下文

至此已经知道，进程的静态描述由进程控制块(PCB)、有关程序段和数据集组成，上面已经介绍了 PCB。本节介绍包括程序段和数据集在内的上下文(context)的概念。

进程上下文实际上是进程执行过程中顺序关联的静态描述。进程上下文是一个与进程切换和处理机状态发生交换有关的概念。在进程执行过程中，由于中断、等待或程序出错等原因造成进程调度，这时操作系统需要知道和记忆进程已经执行到什么地方或新的进程将从何处执行。另外，进程执行过程中还经常出现调用子程序的情况。在调用子程序执行后，进程将返回何处继续执行，执行结果将返回或存放到什么地方等都需要进行记忆。

因此，进程上下文是一个抽象的概念，它包含了每个进程执行过的、执行时的以及待执行的指令和数据，在指令寄存器、堆栈(存放各调用子程序的返回点和参数等)和状态字寄存器等中的内容。

已执行过的进程指令和数据在相关寄存器与堆栈中的内容称为上文，正在执行的指令和数据在寄存器与堆栈中的内容称为正文，待执行的指令和数据在寄存器与堆栈中的内容称为下文。

在不发生进程调度时，进程上下文的改变都是在同一进程内进行的，此时，每条指令的执行对进程上下文的改变较小，一般反映为指令寄存器、程序计数器以及保存调用子程序返回接口用的堆栈值等的变化。

同一进程的上下文的结构由与执行该进程有关的各种寄存器中的值、程序段经过编译后形成的机器指令代码集(或称正文段)、数据集及各种堆栈值与 PCB 结构构成(见图 3.2)。

这里，有关寄存器和栈区的内容是重要的。例如，没有程序计数器 PC 和程序状态字寄存器 PSW，CPU 将无法知道下条待执行指令的地址和控制有关操作。

图 3.3 是 UNIX System Ⅴ的进程上下文组成示例。在 UNIX System Ⅴ中，进程上下文由用户级上下文、寄存器上下文以及系统级上下文组成。用户级上下文由进程的用户程序段部分编译而成的用户正文段、用户数据和用户栈等组成。而寄存器上下文则由程序寄存器(PC)、处理机状态字寄存器(PSW)、栈指针和通用寄存器的值组成。其中，PC 给出 CPU 将要执行的下条指令的虚地址；PSW 给出机器与该进程相关联时的硬件状态，例如当前执行模式、能否执行特权指令等；栈指针指向下一项的当前地址，而通用寄存器则用于不同执行模式之间的参数传递等。

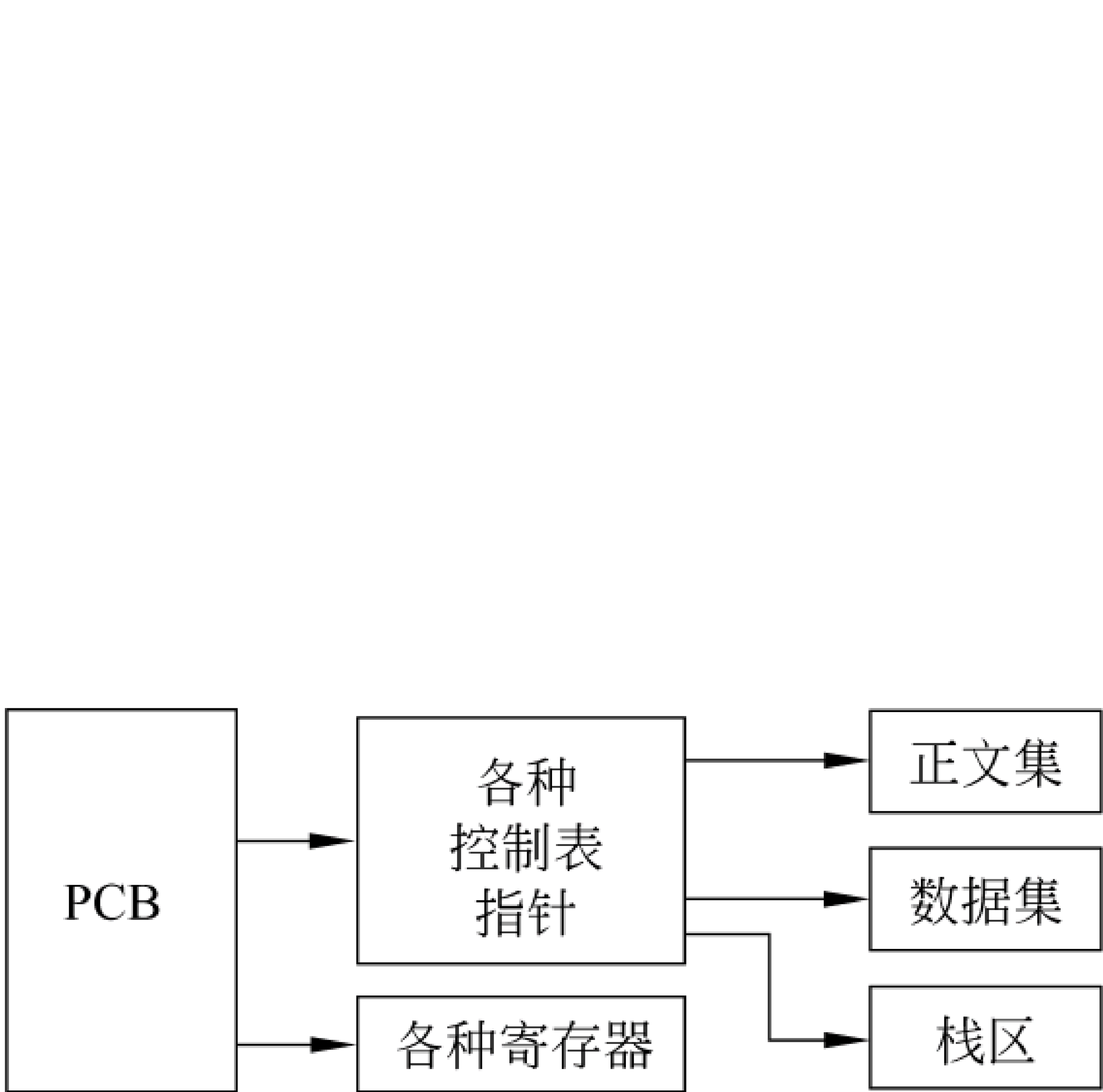


图 3.2 进程上下文结构

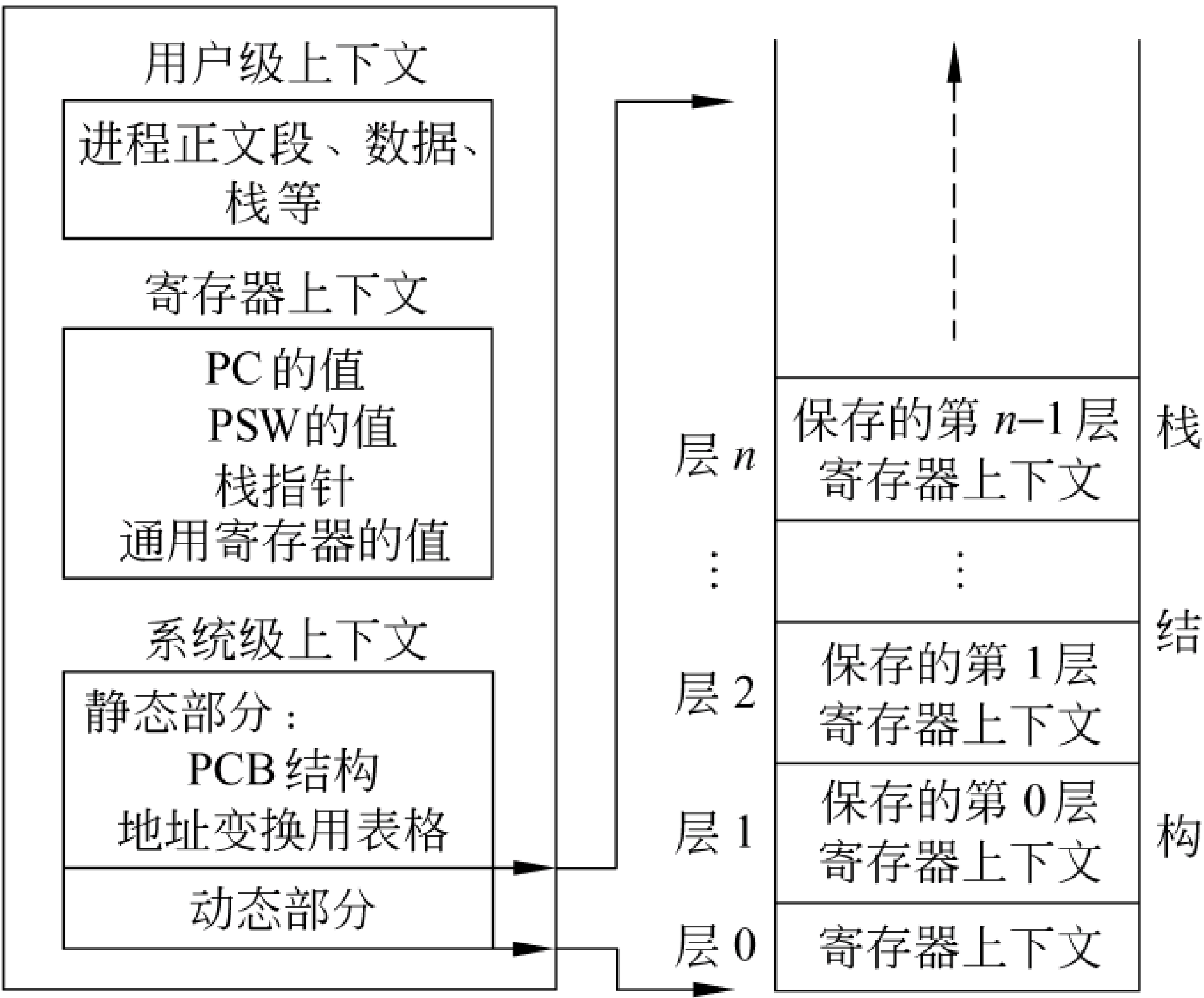


图 3.3 UNIX System Ⅴ进程上下文组成

进程的系统级上下文又分为静态部分与动态部分。这里的动态部分不是指程序的执行，而是指在进入和退出不同的上下文层次时，系统为各层上下文中相关联的寄存器值所保存和恢复的记录。

系统级上下文的静态部分包括 PCB 结构、将进程虚地址空间映射到物理空间用的有关表格和核心栈。这里，核心栈主要用来装载进程中所使用的系统调用的调用序列。

系统级上下文的动态部分是与寄存器上下文相关联的。进程上下文的层次概念也主要

体现在动态部分中，即系统级上下文的动态部分可看成是一些数量变化的层次组成的。其变化规则满足先进后出的堆栈方式，每个上下文层次在栈中各占一项。

### 3.2.3 进程上下文切换

提出进程上下文的概念主要是为了进程上下文的切换。

进程上下文切换发生在不同的进程之间而不是同一个进程内。

进程上下文切换过程一般包含 3 个部分，并涉及 3 个进程。第一部分为保存被切换进程的正文部分（或当前状态）至有关存储区，例如该进程的 PCB 中。第二部分是操作系统进程中有关调度和资源分配程序执行，并选取新的进程。第三部分则是将被选中进程的原来被保存的正文部分从有关存储区中取出，并送至有关寄存器与堆栈中，激活被选中进程执行。

进程上下文切换过程如图 3.4 所示。

进程上下文的切换过程涉及谁来保护和获取进程的正文问题，也就是如何使寄存器和堆栈等中的数据流入流出 PCB 的存储区。

进程上下文切换还涉及系统调度和分配程序，这些都比较耗费 CPU 时间。

为了提高系统执行效率，有的计算机在设计时采用了多组寄存器技术。即进程上下文切换时，不保留被切换进程上下文的正文，但保留切换进程执行时所使用的寄存器。这就减少了数据保存和获取所耗费的时间。

近年来，为了进一步提高执行效率，又提出了线程的概念，本书将在后面的章节中讲述。

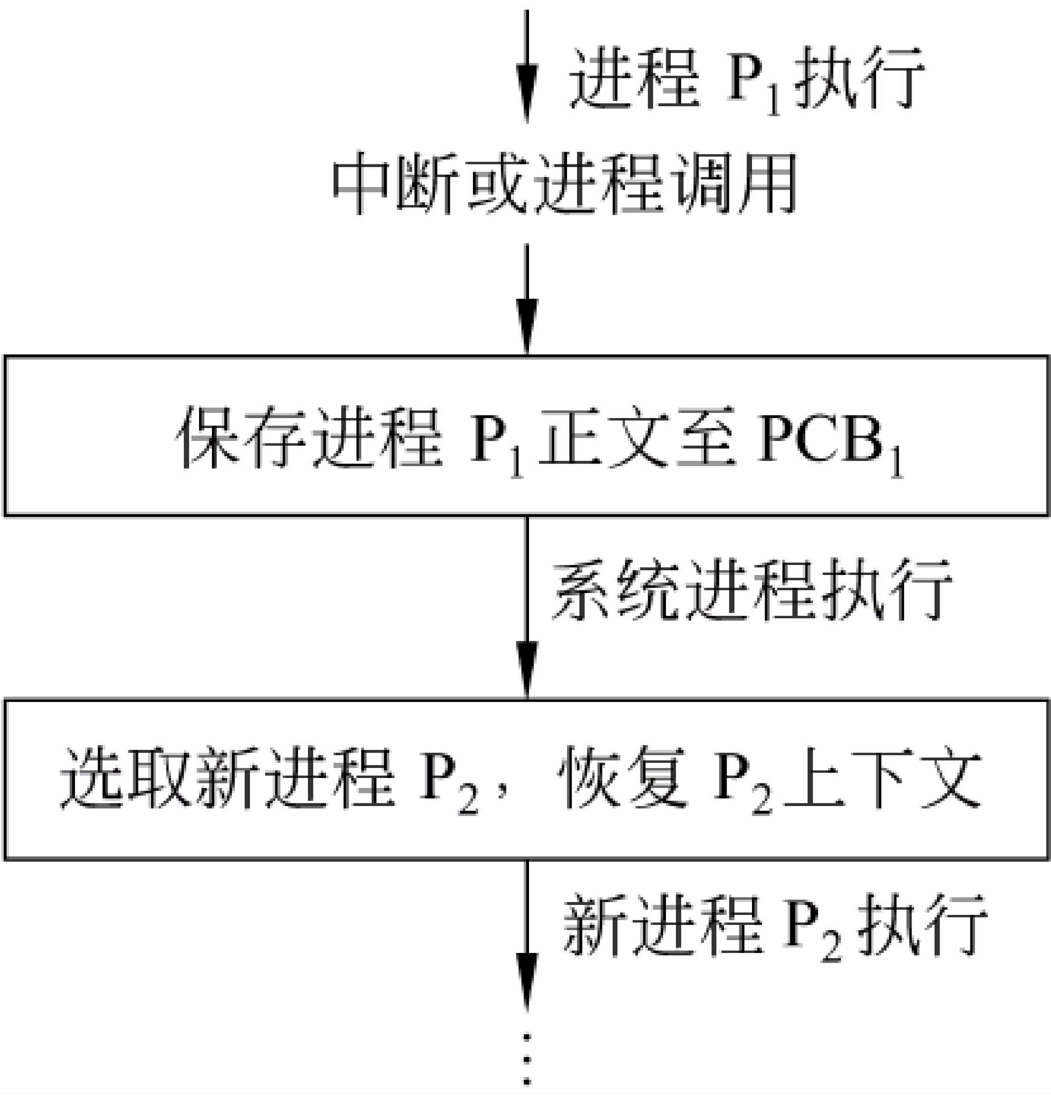



图 3.4 进程上下文的切换过程

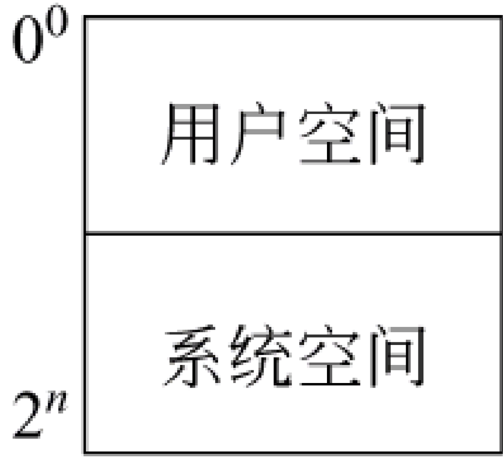



图 3.5 进程空间示例

### 3.2.4 进程空间与大小

任一进程，都有一个自己的地址空间，该空间称为进程空间或虚空间（有关虚空间的概念将在第 5 章中讲述）。进程空间的大小只与处理机的位数有关。例如，一个 16 位长处理机的进程空间大小为 $2^{16}$，而 32 位长处理机的进程空间大小为 $2^{32}$。程序的执行都在进程空间内进行。用户程序、进程的各种控制表格等都按一定的结构排列在进程空间中。另外，在 UNIX 以及 Linux 等操作系统中，进程空间还被划分为用户空间和系统空间两大部分（如图 3.5 所示）。

在进程空间被划分为两大部分后，用户程序在用户空间内执行，而操作系统内核程序则

在系统空间内执行。

另外，为了防止用户程序访问系统空间，造成访问出错，计算机系统还通过程序状态寄存器等设置不同的执行模式，即用户模式和系统模式来进行保护。人们也把用户执行模式和系统执行模式分别称为用户态和系统态。

进程空间将在第 5 章存储管理部分中进一步描述。

进程的大小就是进程空间的大小。

## 3.3 进程状态及其转换

### 3.3.1 进程状态

一个进程的生命期可以划分为一组状态，这些状态刻画了整个进程。系统根据 PCB 结构中的状态值控制进程。前面已介绍过，一个进程在并发执行中，由于资源共享与竞争，有时处于执行状态。有时，进程则因等待某种事件发生而处于等待状态。另外，当一个处于等待状态的进程因等待事件发生被唤醒后，又因不可能立即得到处理机而进入就绪状态。进程刚被创建时，由于其他进程正占有处理机而得不到执行，只能处于初始状态。进程在执行结束后，将退出执行而被终止，这时进程处于终止状态。因此，在进程的生命期内，一个进程至少具有 5 种基本状态：初始态、执行状态、等待状态、就绪状态和终止状态。

处于就绪状态的进程已经得到除 CPU 之外的其他资源，只要由调度得到处理机，便可立即投入执行。

在有些系统中，为了有效地利用内存，就绪状态又可进一步分为内存就绪状态和外存就绪状态。在这样的系统中，只有处于内存就绪状态的进程在得到处理机后才能立即投入执行；而处于外存就绪状态的进程只有先成为内存就绪状态后，才可能被调度执行。这种方式明显地提高了内存的利用效率，但反过来也增加了系统开销和系统复杂性。

在单 CPU 系统中，任一时刻处于执行状态的进程只能有一个。只有处于就绪状态的进程经调度选中之后才可进入执行状态。

在某些操作系统中，一个进程在其生命期内的执行过程中，总要涉及用户程序和操作系统内核程序两部分。因此，进程的执行状态又可进一步划分为用户执行状态（简称为用户态）和系统执行状态（简称为系统态或核心态）。进程的用户程序段在执行时，该进程处于用户态。而一个进程的系统程序段在执行时，该进程处于系统态。为什么要划分用户态和系统态呢？一个最主要的原因是要把用户程序和系统程序区分开来，以利于程序的共享和保护。显然，这也是以增加系统复杂度和系统开销为代价的。

进程因等待某个事件发生而放弃处理机进入等待状态。显然，等待状态可根据等待事件的种类而进一步划分为不同的子状态，例如内存等待、设备等待、文件等待和数据等待等。这样做的好处是系统控制简单，发现和唤醒相应的进程较为容易。但系统中设置过多的状态又会造成系统参数和状态转换过程的增加。

### 3.3.2 进程状态转换

进程的状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化而

转换。那么，是什么样的条件使得进程各状态发生转换呢？图 3.6 给出了 5 个基本状态，即初始状态、就绪状态、执行状态、等待状态与终止状态之间的转换关系。

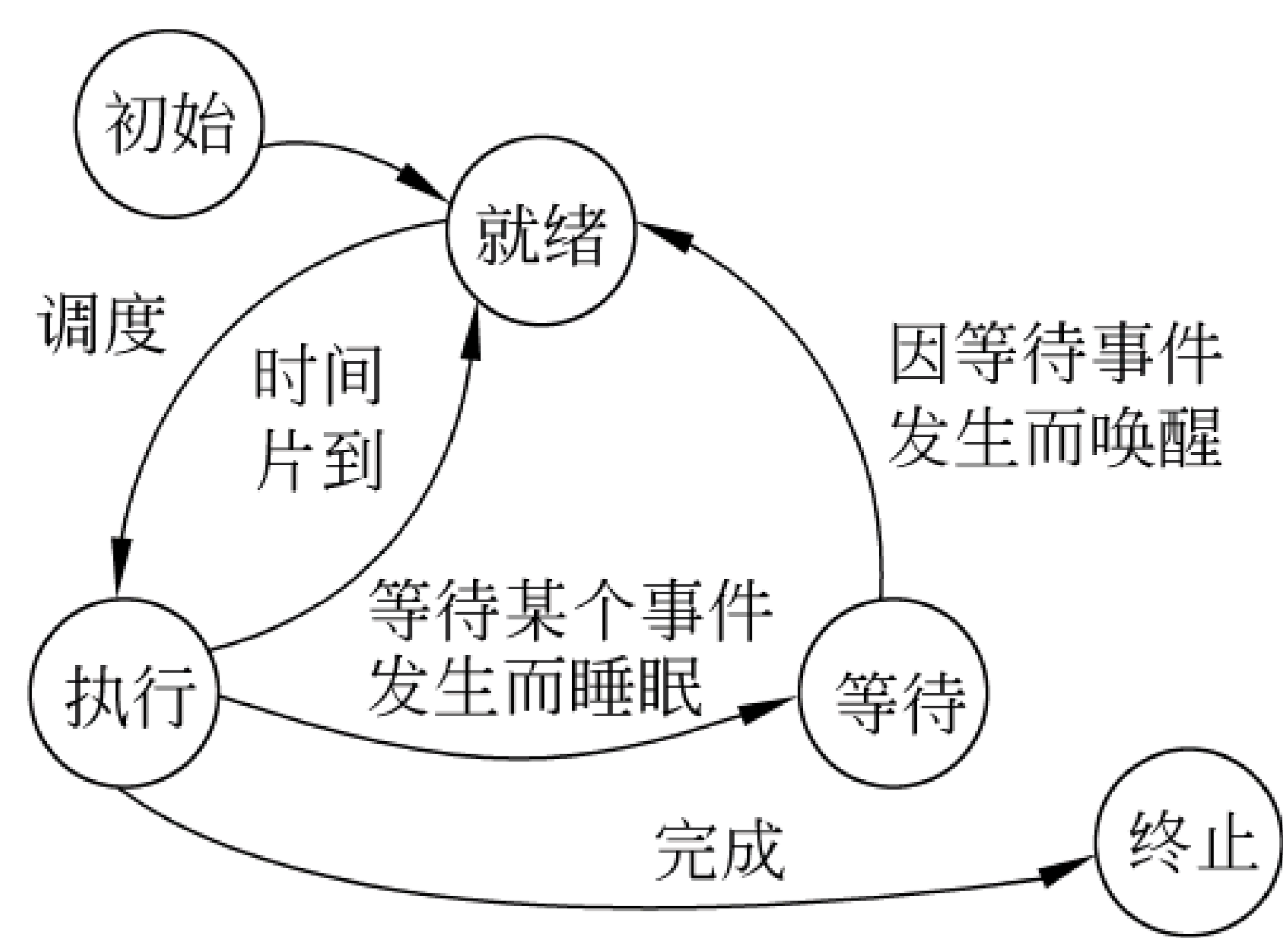



图 3.6　进程状态转换

事实上，进程的状态转换是一个非常复杂的过程。从一个状态到另一个状态的转换除了要使用不同的控制过程（将在 3.4 节中讲述），有时还要借助于硬件触发器才能完成。例如，在 UNIX 系统中，从系统态到用户态的转换要借助硬件触发器完成。

## 3.4　进程控制

进程和处理机管理的一个重要任务是进程控制。所谓进程控制，就是系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各状态间的转换，从而达到多进程高效率并发执行和协调、实现资源共享的目的。一般地，把系统态下执行的某些具有特定功能的程序段称为原语。原语可分为两类：一类是机器指令级的，其特点是执行期间不允许中断，正如在物理学中的原子一样，在操作系统中，它是一个不可分割的基本单位；另一类是功能级的，其特点是作为原语的程序段不允许并发执行。这两类原语都在系统态下执行，且都是为了完成某个系统管理所需要的功能和被高层软件所调用。

显然，系统在创建、撤销一个进程以及要改变进程的状态时，都要调用相应的程序段来完成这些功能。那么，这些程序段是不是原语呢？如果它们不是原语，则由上述原语的定义可知，这些程序段是允许并发执行的。然而，如果不加控制和管理地让这些控制进程状态转换及创建和撤销进程的程序段并发执行，则会使得其执行结果失去封闭性和可再现性（为什么？由读者自答），从而达不到进程控制的目的。反过来，如果对这些程序段采用下面所述的控制方法使其在并发执行过程中也能完成进程控制任务的话，将会大大增加系统的开销和复杂度。因此，在操作系统中，通常把进程控制用程序段做成原语。用于进程控制的原语有创建原语、撤销原语、阻塞原语和唤醒原语等。

### 3.4.1　进程创建与撤销

#### 1. 进程创建

进程创建方式有以下几种。

（1）由系统程序模块统一创建。例如，在批处理系统中，由操作系统的作业调度程序为用户作业创建相应的进程以完成用户作业所要求的功能。

（2）由父进程创建。例如，在层次结构的系统中，父进程创建子进程以完成并行工作。

由系统统一创建的进程之间的关系是平等的，它们之间一般不存在资源继承关系。而父进程与父进程创建的进程之间则存在隶属关系，且互相构成树形结构的家族关系。属于某个家族的一个进程可以继承其父进程所拥有的资源。另外，无论是哪一种方式创建进程，在系统生成时，都必须由操作系统创建一部分承担系统资源分配和管理工作的系统进程。

无论是系统创建方式还是父进程创建方式，都必须调用创建原语来实现。创建原语扫描系统的 PCB 链表，在找到一定的 PCB 表之后，填入调用者提供的有关参数，最后形成代表进程的 PCB 结构。这些参数包括进程名、进程优先级 P0、进程正文段起始地址 d0 和资源清单 R0 等。其实现过程如图 3.7 所示。

**2. 进程撤销**

以下几种情况导致进程被撤销：

(1) 该进程已完成所要求的功能而正常终止；

(2) 由于某种错误导致非正常终止；

(3) 祖先进程要求撤销某个子进程。

无论哪一种情况导致进程被撤销，进程都必须释放它所占用的各种资源和 PCB 结构本身，以利于资源的有效利用。当然，一个进程所占有的某些资源在使用结束时可能早已释放。另外，当一个祖先进程撤销某个子进程时，还需审查该子进程是否还有自己的子孙进程，若有的话，还需撤销其子孙进程的 PCB 结构并释放它们所占有的资源(为什么？请思考)。

撤销原语首先检查 PCB 进程链或进程家族，寻找所要撤销的进程是否存在。如果找到了所要撤销的进程的 PCB 结构，则撤销原语释放该进程所占有的资源之后，把对应的 PCB 结构从进程链或进程家族中摘下并返回给 PCB 空队列。如果被撤销的进程有自己的子进程，则撤销原语先撤销其子进程的 PCB 结构并释放子进程所占用的资源之后，再撤销当前进程的 PCB 结构并释放其资源。撤销原语的实现过程如图 3.8 所示。

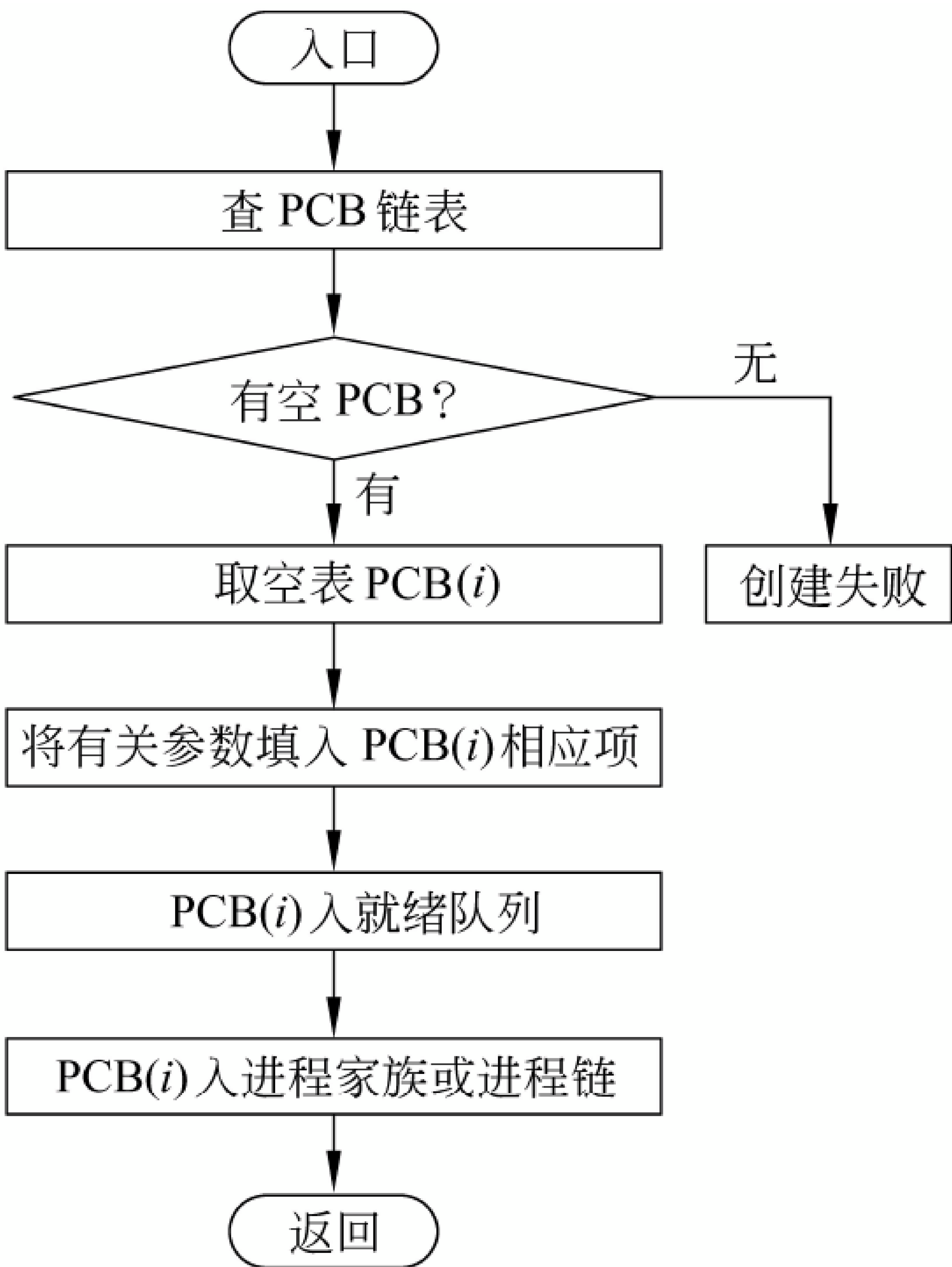



图 3.7 创建原语流图

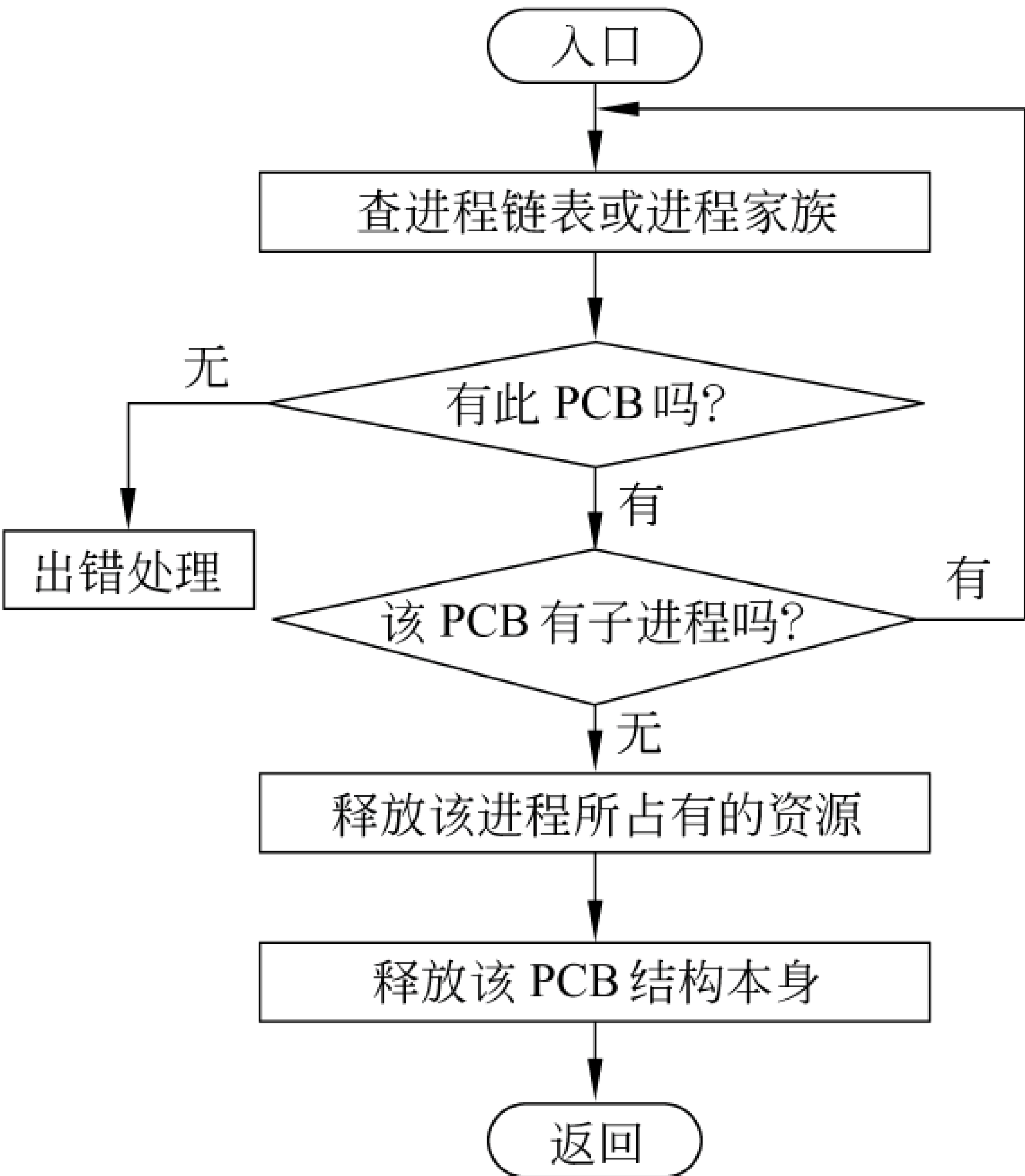



图 3.8 撤销原语流图

### 3.4.2 进程的阻塞与唤醒

进程的创建原语和撤销原语完成了进程从无到有、从存在到消亡的变化。被创建后的进程最初处于就绪状态,然后经调度程序选中后进入执行状态。有关进程调度部分将放在第4章中详述,这里主要介绍实现进程的执行状态到等待状态,又由等待状态到就绪状态转换的两种原语,即阻塞原语与唤醒原语。

阻塞原语在一个进程期待某一事件(例如键盘输入数据、写盘、其他进程发来的数据等)发生,但发生条件尚不具备时,被该进程自己调用来阻塞自己。阻塞原语在阻塞一个进程时,由于该进程正处于执行状态,故应先中断处理机和保存该进程的CPU现场。然后将被阻塞进程置"阻塞"状态后插入等待队列中,再转进程调度程序选择新的就绪进程投入运行。阻塞原语的实现过程如图3.9所示。这里,转进程调度程序是很重要的,否则,处理机将会出现空转而浪费资源。

当等待队列中的进程所等待的事件发生时,等待该事件的所有进程都将被唤醒。显然,一个处于阻塞状态的进程不可能自己唤醒自己(为什么?请思考)唤醒一个进程有两种方法:一种是由系统进程唤醒。另一种是由事件发生进程唤醒。当由系统进程唤醒等待进程时,系统进程统一控制事件的发生并将"事件发生"这一消息通知等待进程。从而使得该进程因等待事件已发生而进入就绪队列。等待进程也可由事件发生进程唤醒。由事件发生进程唤醒时,事件发生进程和被唤醒进程之间是合作关系。因此,唤醒原语既可被系统进程调用,也可被事件发生进程调用。调用唤醒原语的进程称为唤醒进程。唤醒原语首先将被唤醒进程从相应的等待队列中摘下,将被唤醒进程置为就绪状态之后,送入就绪队列。在把被唤醒进程送入就绪队列之后,唤醒原语既可以返回原调用程序,也可以转向进程调度,以便让调度程序有机会选择一个合适的进程执行。唤醒原语的实现框图如图3.10所示。

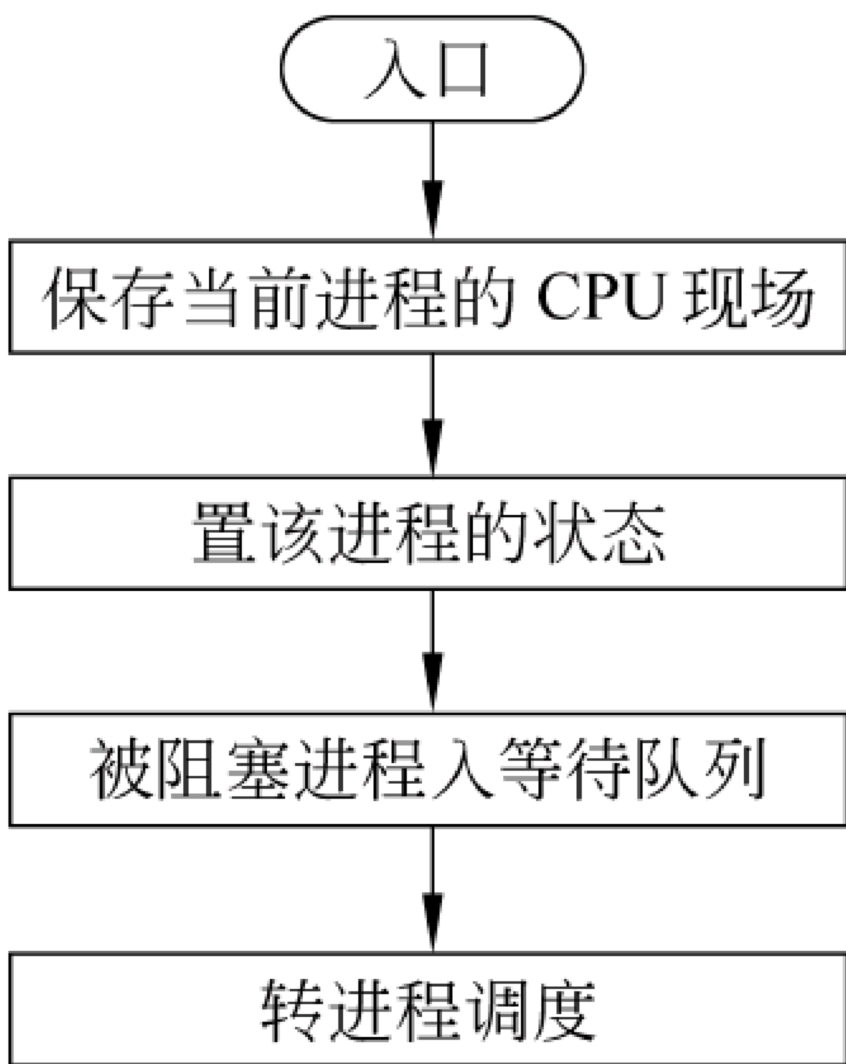



图3.9 阻塞原语

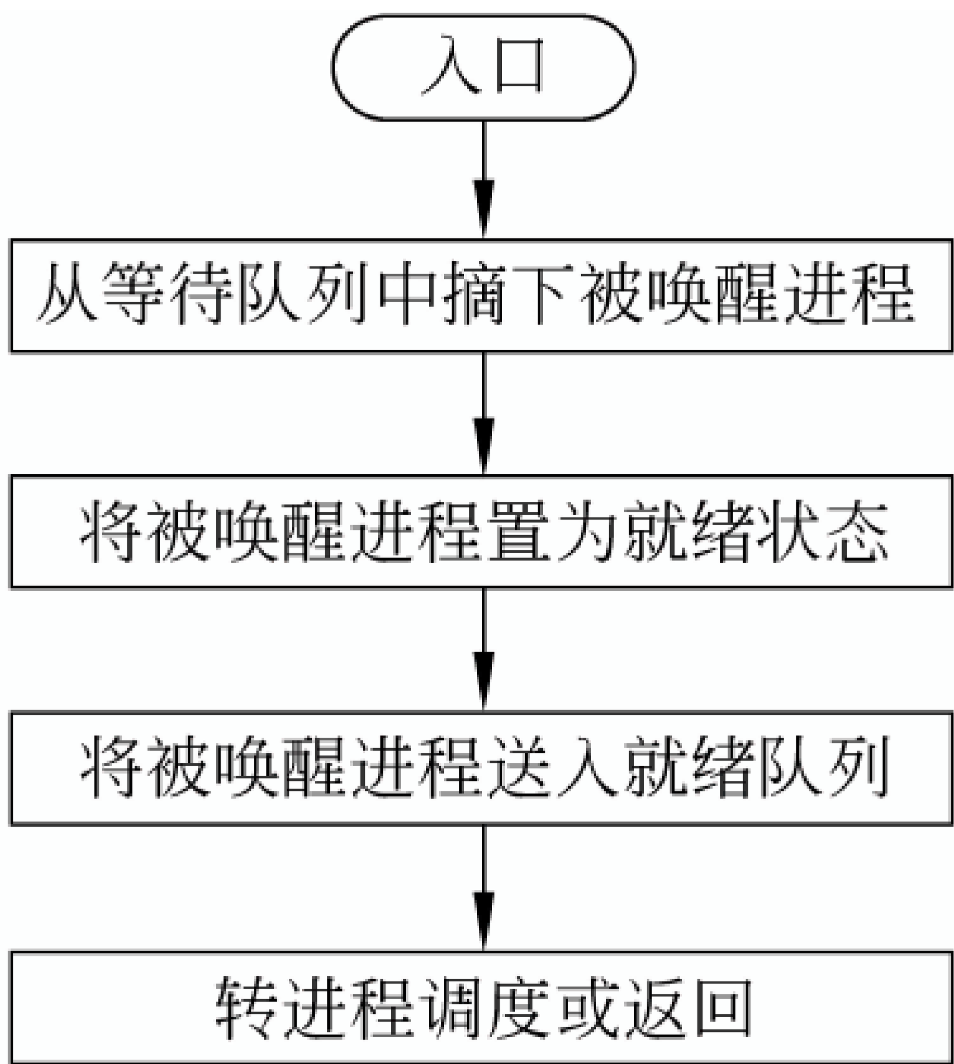



图3.10 唤醒原语

## 3.5 进程互斥

### 3.5.1 资源共享所引起的制约

在介绍进程的概念时,已经讲过进程具有独立性和异步性等并发特征。但是在计算机系统中,由于资源有限,又导致了进程之间的资源竞争和共享。因此,进程的并发执行不仅

仅是用户程序的执行开始时间的随机性和提高资源利用率的结果，也是资源有限性导致资源的竞争与共享对进程的执行过程进行制约所造成的。那么，在进程的并发执行过程中存在哪些制约呢？下面来看这个问题。

**1. 临界区**

在描述一个程序或算法时，总是认为存在一个伪处理机，可以按程序或算法所规定的步骤来执行该程序或算法。但是，事实上，在实际的系统中则往往不是这样，这一点在3.1.1节已介绍。一般说来，即使是程序中所描述的一条语句，也是由多条执行指令构成的。例如，各种程序中经常出现的赋值语句

```
X=X+1;
```

在用汇编语言书写时，就变成

```
LOAD    A,X
ADDI    A,1
STORE   A,X
```

3条语句，这里A代表累加器。根据系统的设计和要求，在这3条语句的执行期间，也有可能发生中断或调度，从而使得与当前进程无关的程序得以执行。为了保证程序执行最终结果的正确性，必须对并发执行的各进程进行制约，以控制它们的执行速度和对资源的竞争。在3.1.2节中已经介绍了进程中两相邻语句可并发执行的3个条件。可是，在实际系统中，要检验即将执行的两相邻语句是否满足这3个条件要花去巨大的系统开销。那么，是否有一种更为简单的办法来检查出需要对程序的哪些部分进行制约才能保证其执行结果的正确性呢？这里来看下面的例子。

设有两个计算进程 $P_A$ 和 $P_B$ 共享内存MS。其中MS分为3个区域，即系统区、进程工作区和数据区。这里数据区被划分为大小相等的块，每个块中既可能放有数据，也有可能未放有数据。系统区主要是堆栈S，其中存放那些空数据块的地址（如图3.11所示）。

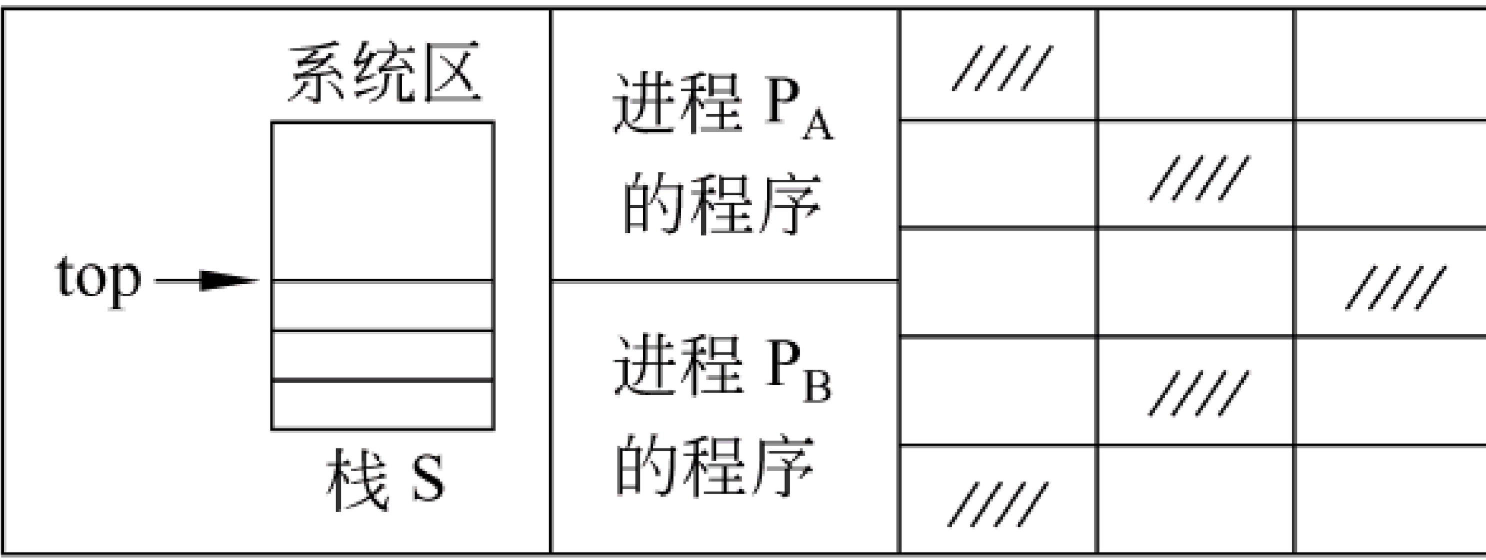



图3.11　多进程共享内存栈区示例

当进程 $P_A$ 或 $P_B$ 要求空数据块时，从堆栈最顶部（top指针所指的位置）取出所需数据块；当进程 $P_A$ 或 $P_B$ 释放数据块时，则把所释放数据块的地址放入堆栈顶部。令getspace为取空数据块过程，release(ad)为释放数据块过程。这里，ad为待释放数据块的地址。如果堆栈S非空的话，进程 $P_A$ 或 $P_B$ 是可以用任意的顺序释放和获取数据块的。执行getspace就是获取一个空数据块，而执行release(ad)就是释放一个地址为ad的数据块。然而，由下面的描述可以看到，在进程并发执行时，getspace或release(ad)将有可能完不成所要求的功能。

getspace和release(ad)可进一步描述为

```
getspace: begin local g
             g←stack [ top ]
             top←top-1
         end
release(ad): begin
             top←top+1
             stack [ top ]←ad
         end
```

设时刻 t0 时，top＝h0，则 getspace 和 release(ad)可能按以下顺序执行：

首先 release(ad)的第一句执行，即

t0：top←top＋1→top＝h0＋1；

接着 getspace 执行，即

t1：g←stack [top]→g＝stack [h0＋1]；

t2：top←top－1→top＝h0；

再是 release(ad)的第二句执行，即

t3：stack [top]←ad→stack [h0]←ad；

其结果是调用 getspace 的进程取到的是 h0＋1 中的一个未定义值，而调用 release(ad)的进程把所释放的空块地址 ad 重新放入了 h0 中。

怎样保证上述执行结果的正确性呢？一个较为明显的答案是，如果把 getspace 和 release(ad)抽象为两个各以一个动作完成的顺序执行单位，那么执行结果的正确性是可以保证的。

把不允许多个并发进程交叉执行的一段程序称为临界部分(critical section)或临界区(critical region)。

临界区是由属于不同并发进程的程序段共享公用数据或公用数据变量而引起的，例如上例中就是因为过程 getspace 和 release(ad)共同访问栈 S 中的数据而引起的。临界区不可能用增加硬件的方法来解决。因此，临界区也可以被称为访问公用数据的那段程序。

**2. 间接制约**

一般来说，可以把那些不允许交叉执行的临界区按不同的公用数据划分为不同的集合。在上例中，以公用数据栈 S 划分的临界区集合是{getspace，release}。把这些集合称为类(class)。显然，对类给定一个唯一的标识名，系统就会容易地区分它们。在程序的描述中，可用下列标准形式来描述临界区：

```
when<类名>do<临界区>od
```

设类{getspace，release}的类名为 sp，则 getspace 和 release(ad)可重新描述为：

```
getspace: when sp do getspce←stack [ top ]
                top←top-1 od
release(ad): when sp do top← top+1
                stack [ top ]←ad od
```

把这种由于共享某一公有资源而引起的在临界区内不允许并发进程交叉执行的现象，称为由共享公有资源而造成的对并发进程执行速度的间接制约，简称间接制约。这里，“间

接”二字主要是指各并发进程的速度受公有资源制约，而不是进程间直接制约的意思。

这里，受间接制约的类中各程序段在执行顺序上是任意的。

显然，对于每一类，系统应有相应的分配和释放相应公有资源的管理办法，以制约并发进程。这就是互斥。

**3. 什么是互斥**

综上所述，可以把互斥定义为：一组并发进程中的一个或多个程序段，因共享某一公有资源而导致它们必须以一个不允许交叉执行的单位执行。也就是说，不允许两个以上的共享该资源的并发进程同时进入临界区称为互斥。

这里，考虑类中只有一个元素，也就是只有一个程序段的情况是很有意思的。这时程序段本身为公有资源，被并发进程共享。一般情况下，作为程序段的一个过程不允许多个进程同时访问它。但如果该过程是纯过程，则各并发进程可以同时访问它。纯过程是指在执行过程中不改变过程自身代码的一类过程。通常，在计算机系统中，有许多过程是被多个并发进程共享，例如编辑程序、编译程序等。把一个过程作成纯过程可便于多个进程共享，但由于编制纯过程必须对有关变量和工作区作相应的处理，从而使其执行效率往往会受到一定的影响。

一组并发进程互斥执行时必须满足如下准则：

(1) 不能假设各并发进程的相对执行速度。即各并发进程享有平等地、独立地竞争共有资源的权利，且在不采取任何措施的条件下，在临界区内任一指令结束时，其他并发进程可以进入临界区。

(2) 并发进程中的某个进程不在临界区时，它不阻止其他进程进入临界区。

(3) 并发进程中的若干个进程申请进入临界区时，只能允许一个进程进入。

(4) 并发进程中的某个进程从申请进入临界区时开始，应在有限时间内得以进入临界区。

这里，准则(1)、(2)、(3)是保证各并发进程享有平等地、独立地竞争和使用公有资源的权利，且保证每一时刻至多只有一个进程在临界区。而准则(4)则是并发进程不发生死锁(将在后面讲述)的重要保证。否则，由于某个并发进程长期占有临界区，其他进程则因为不能进入临界区而进入互相等待状态。

在一组并发执行进程中，除了因为竞争公有资源而引起的间接制约带来进程之间互斥之外，还存在因为并发进程互相共享对方的私有资源所引起的直接制约。直接制约将使得各并发进程同步执行。有关直接制约与进程间的同步将在后续章节中讨论。下面，将讨论互斥的实现方法。

### 3.5.2 互斥的加锁实现

3.5.1 节中给出了临界区的描述方法和并发进程互斥执行时所必须遵守的准则。但是，并没有给出怎样实现并发进程的互斥。人们可能认为只需把临界区中的各个过程按不同的时间排列调用就行了。但事实上这是不可能的。因为这要求该组并发进程中的每个进程事先知道其他并发进程与系统的动作，由用户程序执行开始的随机性可知，这是不可能的。

一种可能的办法是对临界区加锁以实现互斥。当某个进程进入临界区之后，它将锁上临界区，直到它退出临界区时为止。并发进程在申请进入临界区时，首先测试该临界区是否

是上锁的。如果该临界区已被锁住,则该进程要等到该临界区开锁之后才有可能获得临界区。设临界区的类名为 S。为了保证每一次临界区中只能有一个程序段被执行,又设锁定位为 key[S],它表示该锁定位属于类名为 S 的临界区。加锁后的临界区程序描述如下:

```
lock(key[S])
<临  界  区>
unlock(key[S])
```

设 key[S]=1 时表示类名为 S 的临界区可用,key [S]=0 时表示类名为 S 的临界区不可用。则,unlock(key [S])只用一条语句即可实现,即

```
key[S]←1
```

不过,由于 lock(key [S])必须满足 key [S]=0 时不允许任何进程进入临界区,而 key [S]=1 时仅允许一个进程进入临界区的准则,因而实现起来较为困难。

一种简便的实现方法是

```
lock (x): begin local v
             repeat
                 v←x
             until v=1
             x←0
     end
```

不过,这种实现方法是不能保证并发进程互斥执行所要求的准则(3)的。因为当同时有几个进程调用 lock(key[S])时,在 x←0 语句执行之前,可能已有两个以上的进程由于 key [S]=1 而进入临界区。为了解决这个问题,有些计算机在硬件中设置了"测试与设置"(test and set)指令,从而保证第一步和第二步执行的不可分离性。

这里,有一点需要注意的是:在系统实现时锁定位 key[S]总是设置在公有资源所对应的数据结构中的。

### 3.5.3 信号量和 P、V 原语

**1. 信号量(semaphore)**

尽管用加锁的方法可以实现进程之间的互斥,但这种方法仍然存在一些影响系统可靠性和执行效率的问题。例如,循环测试锁定位将损耗较多的 CPU 计算时间。如果一组并发进程的进程数较多,且由于每个进程在申请进入临界区时都得对锁定位进行测试,这种开销是很大的。

另外,使用加锁法实现进程间互斥时,还将导致在某些情况下出现不公平现象。试考虑以下进程 $P_A$ 和 $P_B$ 反复使用临界区的情况。

进程 $P_A$:

```
A: lock(key[S])
      <S>
   unlock(key[S])
   Goto A
```

进程 $P_B$：

```
B: lock(key[S])
     <S>
   unlock(key[S])
   Goto B
```

设进程 $P_A$ 已通过 lock(key[S])过程而进入临界区。显然，在进程 $P_A$ 执行 unlock(key[S])过程之前，key[S]=0 且进程 $P_B$ 没有进入临界区的机会。然而，当进程 $P_A$ 执行完 unlock(key[S])过程之后，由于紧接着是一条转向语句，进程 $P_A$ 将又立即去执行 lock(key[S])过程。此时，由于 unlock(key[S])过程已将 key[S]的值置为 1，也就是开锁状态，从而进程 $P_A$ 又可进入临界区 S。只有在进程 $P_A$ 执行完 unlock(key[S])过程之后、执行 Goto A 语句之前的瞬间发生进程调度，使进程 $P_A$ 把处理机转让给进程 $P_B$，进程 $P_B$ 才有可能得到执行。然而遗憾的是，这种可能性是非常小的。因此，进程 $P_B$ 将处于永久饥饿状态(starvation)。

怎样解决上述问题呢？首先，必须找到产生上述问题的原因。显然，在用加锁法解决进程互斥的问题时，一个进程能否进入临界区是依靠进程自己调用 lock 过程去测试相应的锁定位。也就是说，每个进程能否进入临界区是依靠自己的测试判断。这样，没有获得执行机会的进程当然无法判断，从而出现不公平现象。而获得了测试机会的进程又因需要测试而损失一定的 CPU 时间。这正如某个学生想使用某个人人都可借用，且不规定使用时间的教室一样，他必须首先申请获得使用该教室的权利，然后再到教室看看该教室是不是被锁上了。如果该教室被锁上了，他只好下次再来观察，看该教室的门是否已被打开。这种反复将持续到他进门为止。从这个例子中，可以得到解决加锁法所带来的问题的方法。一种最直观的办法是，设置一个教室管理员。从而，如果有学生申请使用教室而未能如愿时，教室管理员记下他的名字，并等到教室门一打开就通知该学生进入。这样，既减少了学生多次来去教室检查门是否被打开的时间，又减少了因为学生自发地检查造成的不公平现象(有的学生可能来几十次也进不了教室门，但有的学生可能一次就进去了，或不断地出出进进)。在操作系统中，这个管理员就是信号量。信号量管理相应临界区的公有资源，它代表可用资源实体。

信号量的概念和下面所述的 P、V 原语是荷兰科学家 E. W. Dijkstra 提出来的。信号是铁路交通管理中的一种常用设备，交通管理人员利用信号颜色的变化来实现交通管理。在操作系统中，信号量 sem 是一个整数。在 sem 大于等于零时代表可供并发进程使用的资源实体数，但 sem 小于零时则表示正在等待使用临界区的进程数。显然，用于互斥的信号量 sem 的初值应该大于零，而建立一个信号量必须说明所建信号量代表的意义，赋初值，以及建立相应的数据结构，以便指向那些等待使用该临界区的进程。

**2. P、V 原语**

信号量的数值仅能由 P、V 原语操作改变(P 和 V 分别是荷兰语 Passeren 和 Verhoog 的头一个字母，相当于英文的 pass 和 increment 的意思)。采用 P、V 原语，可以把类名为 S 的临界区描述为 When S do P(sem)临界区 V(sem)od。

这里，sem 是与临界区内所使用的公用资源有关的信号量。一次 P 原语操作使得信号量 sem 减 1，而一次 V 原语操作将使得信号量 sem 加 1。必须强调的一点是，当某个进程正

在临界区内执行时,其他进程如果执行了 P 原语操作,则该进程并不像调用 lock 时那样因进不了临界区而返回到 lock 的起点,等以后重新执行测试,而是在等待队列中等待有其他进程做 V 原语操作释放资源后,进入临界区,这时,P 原语的执行才算真正结束。另外,当有好几个进程执行 P 原语未通过而进入等待状态之后,如有某进程执行了 V 原语操作,则等待进程中的一个可以进入临界区,但其他进程必须等待。

P 原语操作的主要动作如下:

(1) sem 减 1。

(2) 若 sem 减 1 后仍大于或等于零,则 P 原语返回,该进程继续执行。

(3) 若 sem 减 1 后小于零,则该进程被阻塞后进入与该信号相对应的队列中,然后转进程调度。

P 原语操作的功能框图如图 3.12 所示。

V 原语的操作主要动作如下:

(1) sem 加 1。

(2) 若相加结果大于零,V 原语停止执行,该进程返回调用处,继续执行。

(3) 若相加结果小于或等于零,则从该信号的等待队列中唤醒一个等待进程,然后再返回原进程继续执行或转进程调度。

V 原语操作的功能框图如图 3.13 所示。

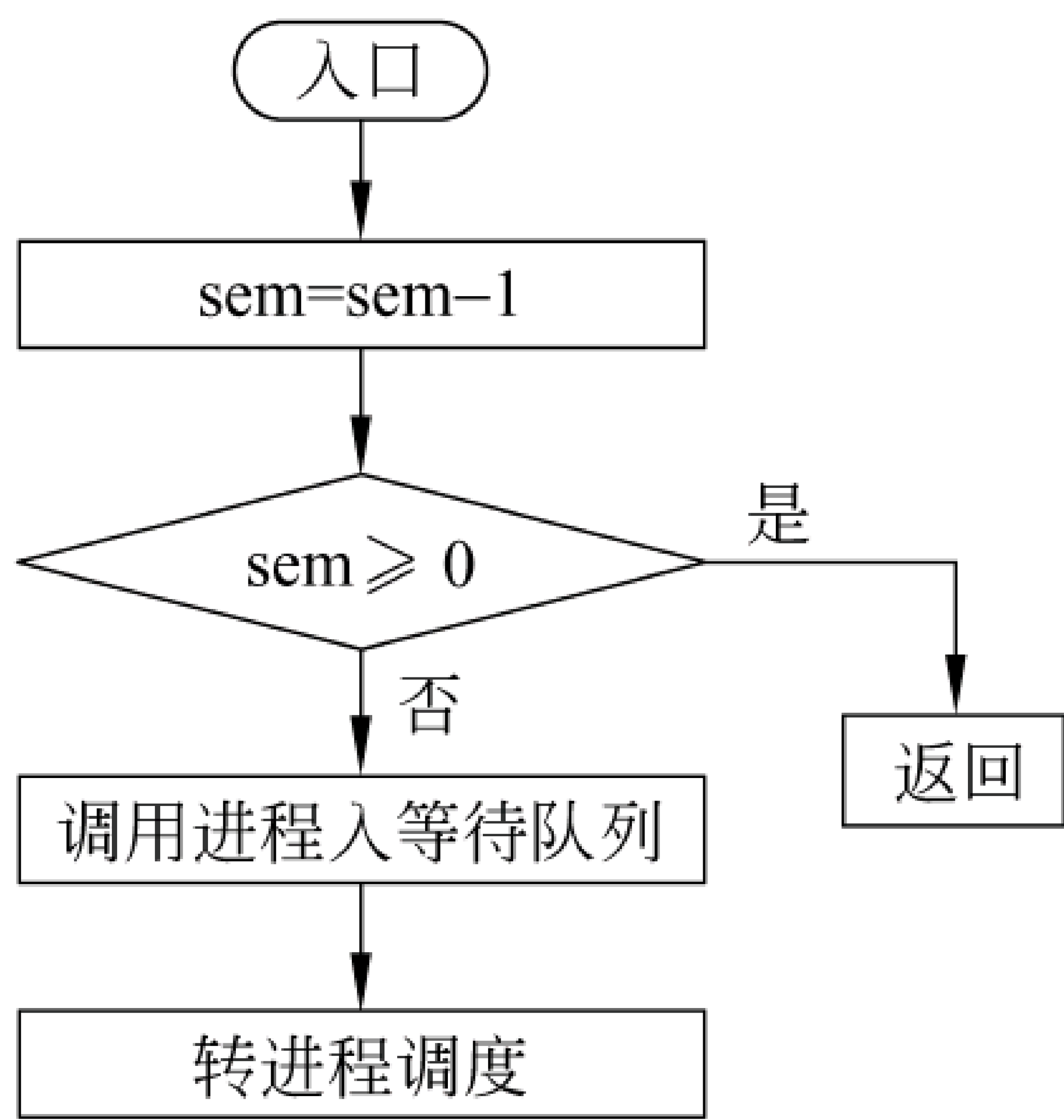



图 3.12　P 原语操作功能

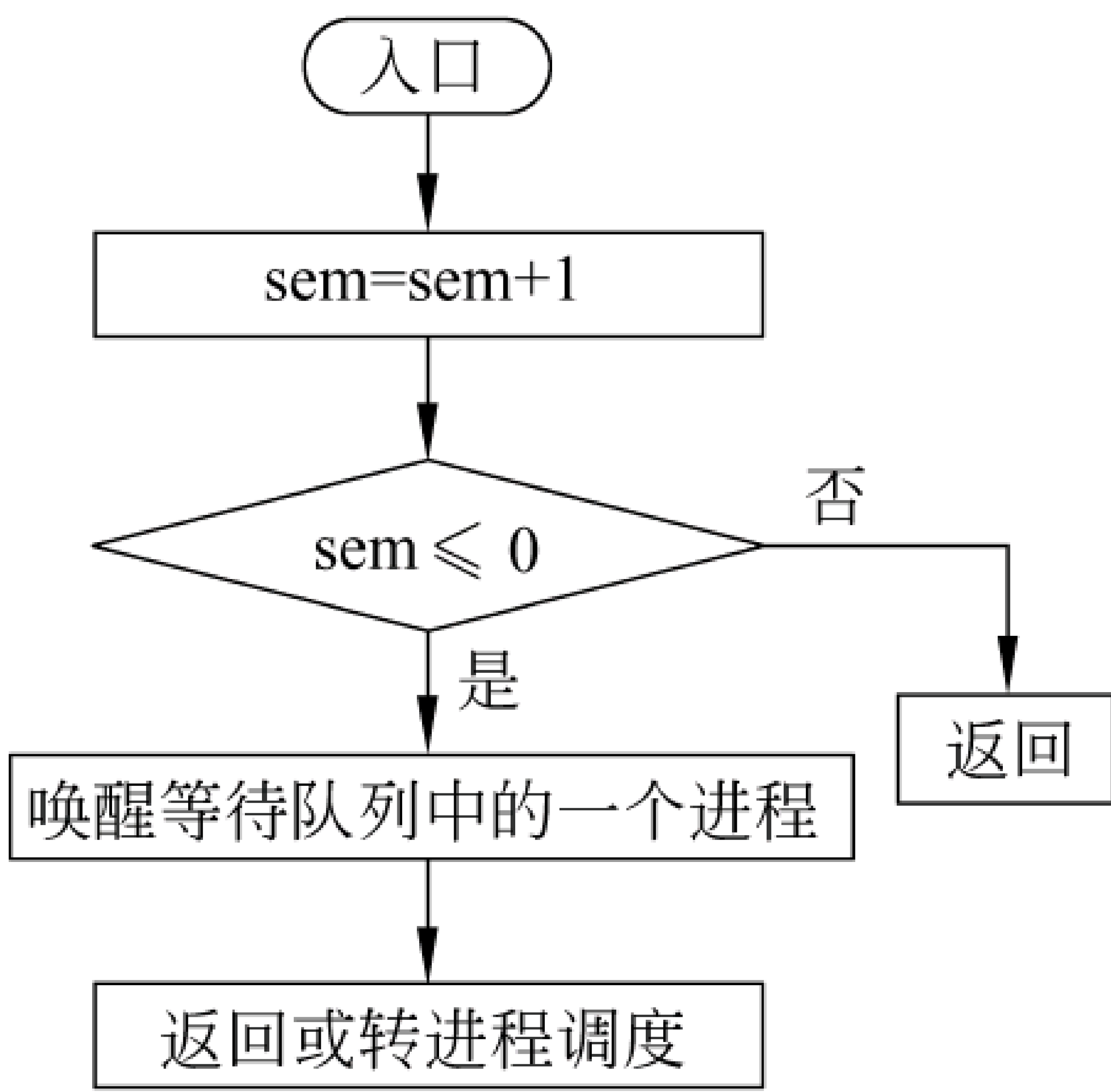



图 3.13　V 原语操作功能

有了加锁法的基础,大家应该明白为什么 P、V 过程要以原语实现的原因。否则,如果多个进程同时调用 P 操作或 V 操作的话,则有可能在 P 操作刚执行完 sem−1 而未把对应进程送入等待队列时,V 操作开始执行。从而,V 操作将无法发现等待进程而返回。因此,P、V 操作都必须以原语实现,且在 P、V 原语执行期间不允许中断发生。

关于 P、V 原语的实现,有许多方法。这里介绍一种使用加锁法的软件实现方法,其实现过程描述如下:

```
P(sem):
    begin
        封锁中断;
        lock(lockbit)
        val[sem]=val[sem]-1
        if val[sem]<0
```

```
            保护当前进程 CPU 现场
            当前进程状态置为"等待"
            将当前进程插入信号 sem 等待队列
            转进程调度
        fi
        unlock(lockbit);开放中断
    end
V(sem):
    begin
        封锁中断;
        lock(lockbit)
        val[sem]=val[sem]+1
        if val[sem]≤0
            local k
            从 sem 等待队列中选取一个等待进程,将其指针置入 k 中
            将 k 插入就绪队列
            进程状态置为"就绪"
        fi
        unlock(lockbit);开放中断
    end
```

### 3.5.4　用 P、V 原语实现进程互斥

利用 P、V 原语和信号量,可以方便地解决并发进程的互斥问题,而且不会产生使用加锁法解决互斥问题时所出现的问题。

设信号量 sem 是用于互斥的信号量,且其初值为 1 表示没有并发进程使用该临界区。显然,由 3.5.3 节的讨论可知,只要把临界区置于 P(sem)和 V(sem)之间,即可实现进程间的互斥。当一个进程想要进入临界区时,它必须先执行 P 原语操作以将信号量 sem 减 1。在一个进程完成对临界区的操作之后,它必须执行 V 原语操作以释放它所占用的临界区。由于信号量初始值为 1,所以,任一进程在执行 P 原语操作之后将 sem 的值变为 0,表示该进程可以进入临界区。在该进程未执行 V 原语操作之前如有另一进程想进入临界区的话,它也应先执行 P 原语操作,从而使 sem 的值变为－1,因此,第二个进程将被阻塞。直到第一个进程执行 V 原语操作之后,sem 的值变为 0,从而可唤醒第二个进程进入就绪队列,经调度后再进入临界区。在第二个进程执行完 V 原语操作之后,如果没有其他进程申请进入临界区的话,则 sem 又恢复到初始值。

用信号量实现两个并发进程 $P_A$ 和 $P_B$ 互斥的描述如下:

(1) 设 sem 为互斥信号量,其取值范围为(1,0,－1)。其中 sem＝1 表示进程 $P_A$ 和 $P_B$ 都未进入类名为 S 的临界区,sem＝0 表示进程 $P_A$ 或 $P_B$ 已进入类名为 S 的临界区,sem＝－1 表示进程 $P_A$ 和 $P_B$ 中,一个进程已进入临界区,而另一个进程等待进入临界区。

(2) 实现过程描述如下:

```
PA:
    P(sem)
    <S>
```

```
        V(sem)
        ⋮
    P_B:
        P(sem)
        <S>
        V(sem)
        ⋮
```

# 3.6 进 程 同 步

## 3.6.1 同步的概念

3.5 节中，由并发进程同时访问公有数据和公有变量等对公有资源的竞争，引出了进程互斥的概念以及互斥的实现方法。那么，除了对公有资源的竞争而引起的间接制约之外，并发进程之间是否还存在着其他制约关系影响执行速度呢？现在来看下面的例子。

计算进程和打印进程共同使用同一缓冲区 Buf。计算进程反复地把每次计算结果放入 Buf 中，而打印进程则把计算进程每次放入 Buf 中的数据通过打印机打印输出。如果不采取任何制约措施，这两个进程的执行起始时间和执行速度都是彼此独立的，其相应的控制段可以描述如下：

```
    P_C:
        A:local Buffer
            repeat
                Buffer← Buf
            until Buf=空
            计算
            得到计算结果
            Buf←计算结果
            goto A
    P_P:
        B:local Pri
            repeat
                Pri←Buf
            until Pri≠空
            打印 Buf 中的数据
            清除 Buf 中的数据
            goto B
```

这里，如果假定进程 $P_C$ 和 $P_P$ 对公用缓冲区 Buf 已采取了互斥措施。

显然，如果按上面的描述并发执行进程 $P_C$ 和 $P_P$ 的话，则会造成 CPU 时间的极大浪费(因为其中包含两处反复测试语句)。这是操作系统设计要求不允许的。CPU 时间的浪费主要是由于进程 $P_C$ 和 $P_P$ 的执行互相制约所引起的。$P_C$ 的输出结果是 $P_P$ 的执行条件，反过来，$P_P$ 的执行结果也是 $P_C$ 的执行条件。这种现象在操作系统和用户进程中大量存在。这与 3.5 节中讲述的进程互斥是不同的，进程互斥时它们的执行顺序可以是任意的。一组

在异步环境下的并发进程，各自的执行结果互为对方的执行条件，从而限制各进程的执行速度的过程称为并发进程间的直接制约。这里异步环境主要指各并发进程的执行起始时间的随机性和执行速度的独立性。正如在上面例子中所看到的那样，如果没有相应的解决方法，进程的直接制约将会造成大量的 CPU 时间浪费。一种最为简单和直观的方法是直接制约的进程互相给对方进程发送执行条件已经具备的信号。这样，被制约进程即可省去对执行条件的测试，只要收到了制约进程发来的信号便开始执行，而在未收到制约进程发来的信号时便进入等待状态。

把异步环境下的一组并发进程因直接制约而互相发送消息而进行互相合作、互相等待，使得各进程按一定的速度执行的过程称为进程间的同步。具有同步关系的一组并发进程称为合作进程，合作进程间互相发送的信号称为消息或事件。如果对一个消息或事件赋予唯一的消息名，则可用过程

```
wait(消息名)
```

表示进程等待合作进程发来的消息，而用过程

```
signal(消息名)
```

表示向合作进程发送消息。利用过程 wait 和 signal，可以简单地描述上面例子中的计算进程 $P_C$ 和打印进程 $P_P$ 的同步关系如下：

（1）设消息名 Bufempty 表示 Buf 空，消息名 Buffull 表示 Buf 中装满了数据。

（2）初始化 Bufempty＝true，Buffull＝false。

（3）描述如下：

```
Pc:
    A: wait(Bufempty)
        计算
        Buf←计算结果
        Bufempty←false
        signal(Buffull)
        goto A
Pp:
    B: wait(Buffull)
        打印 Buf 中的数据
        清除 Buf 中的数据
        Buffull←false
        signal(Bufempty)
        goto B
```

过程 wait 的功能是等待到消息名为 true 的进程继续执行，而 signal 的功能则是向合作进程发送合作进程所需要的消息名，并将其值置为 true。

### 3.6.2 私用信号量

上面用 wait(消息名)与 signal(消息名)的方式描述了进程同步的一种实现方法。事实上，使用 3.5 节介绍的信号量的方法也可实现进程间的同步。

一般来说，也可以把各进程之间发送的消息作为信号量看待。与进程互斥时不同的是，这里的信号量只与制约进程及被制约进程有关而不是与整组并发进程有关。因此，称该信号量为私用信号量(private semaphore)。一个进程 $P_i$ 的私用信号量 $Sem_i$ 是从制约进程发送来的进程 $P_i$ 的执行条件所需要的消息。与私用信号量相对应，称互斥时使用的信号量为公用信号量。

### 3.6.3 用 P、V 原语操作实现同步

有了私用信号量的概念，可以使用 P、V 原语操作实现进程间的同步，其实现方法与利用 wait 和 signal 过程时相同，也是分为 3 步。首先为各并发进程设置私用信号量，然后为私用信号量赋初值，最后利用 P、V 原语和私用信号量规定各进程的执行顺序。

**例**：设进程 $P_A$ 和 $P_B$ 通过缓冲区队列传递数据(如图 3.14 所示)。$P_A$ 为发送进程，$P_B$ 为接收进程。$P_A$ 发送数据时调用发送过程 deposit(data)，$P_B$ 接收数据时调用过程 remove(data)，且数据的发送和接收过程满足如下条件：

(1) 在 $P_A$ 至少送一块数据入一个缓冲区之前，$P_B$ 不可能从缓冲区中取出数据(假定数据块长等于缓冲区长度)。

(2) $P_A$ 往缓冲队列发送数据时，至少有一个缓冲区是空的。

(3) 由 $P_A$ 发送的数据块在缓冲队列中按先进先出(FIFO)方式排列。

描述发送过程 deposit(data)和接收过程 remove(data)。




图 3.14　缓冲区队列

**解**：由题意可知，进程 $P_A$ 调用的过程 deposit(data)和进程 $P_B$ 调用的过程 remove(data)必须同步执行，因为过程 deposit(data)的执行结果是过程 remove(data)的执行条件，而当缓冲队列全部装满数据时，remove(data)的执行结果又是 deposit(data)的执行条件，满足同步定义。从而，按以下 3 步描述过程 deposit(data)和 remove(data)：

(1) 设 Bufempty 为进程 $P_A$ 的私用信号量，Buffull 为进程 $P_B$ 的私用信号量。

(2) 令 Bufempty 的初始值为 $n$($n$ 为缓冲队列的缓冲区个数)，Buffull 的初始值为 0。

(3) 实现过程描述如下：

```
PA: deposit(data):
    begin local x
        P(Bufempty);
        按 FIFO 方式选择一个空缓冲区 Buf(x)
        Buf(x)← data
        Buf(x)置满标记
        V(Buffull)
    end
PB: remove(data):
    begin local x
        P (Buffull);
        按 FIFO 方式选择一个装满数据的缓冲区 Buf(x)
```

```
        data←Buf(x)
        Buf(x)置空标记
        V (Bufempty)
    end
```

这里，局部变量 x 用来指明缓冲区的区号，给 Buf(x)置标志位是为了便于区别和搜索空缓冲区及非空缓冲区。(思考：在该题中需要考虑互斥吗？为什么？如果每次只允许一个进程对缓冲队列进行操作时怎么办?)

### 3.6.4 生产者-消费者问题

把并发进程的同步和互斥问题一般化，可以得到一个抽象的一般模型，即生产者-消费者问题(producer-consumer problems)。计算机系统中，每个进程都申请使用和释放各种不同类型的资源，这些资源既可以是外设、内存及缓冲区等硬件资源，也可以是临界区、数据和例程等软件资源。把系统中使用某一类资源的进程称为该资源的消费者，而把释放同类资源的进程称为该资源的生产者。例如，在上面的计算进程 $P_C$ 与打印进程 $P_P$ 公用一个缓冲区的例子中，计算进程 $P_C$ 把数据送入缓冲区，打印进程 $P_P$ 从缓冲区中取数据打印输出，因此，$P_C$ 进程相当于数据资源的生产者，而 $P_P$ 进程相当于消费者。

把一个长度为 $n$ 的有界缓冲区($n>0$)与一群生产者进程 $P_1, P_2, \cdots, P_m$ 和一群消费者进程 $C_1, C_2, \cdots, C_k$ 联系起来(如图 3.15 所示)。

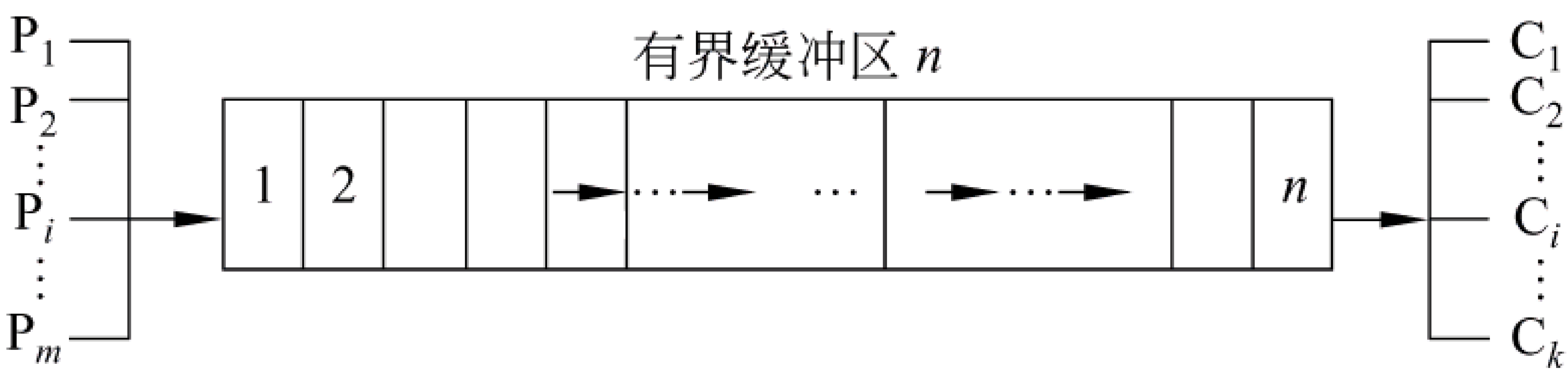



图 3.15 生产者-消费者问题

设生产者进程和消费者进程是互相等效的，其中，各生产者进程使用的过程 deposit (data)和各消费者使用的过程 remove(data)可描述如下：

首先，可以看到，上述生产者-消费者问题是一个同步问题。即生产者和消费者之间满足如下条件：

(1) 消费者想接收数据时，有界缓冲区中至少有一个单元是满的。

(2) 生产者想发送数据时，有界缓冲区中至少有一个单元是空的。

另外，由于有界缓冲区是临界资源，因此，各生产者进程和各消费者进程之间必须互斥执行。

由以上分析，设公用信号量 mutex 保证生产者进程和消费者进程之间的互斥，设信号量 avail 为生产者进程的私用信号量，信号量 full 为消费者进程的私用信号量。信号量 avail 表示有界缓冲区中的空单元数，初值为 $n$；信号量 full 表示有界缓冲区中非空单元数，初值为 0。信号量 mutex 表示可用有界缓冲区的个数，初值为 1。从而有

```
deposit(data):
    begin
        P(avail)
```

```
            P(mutex)
            送数据入缓冲区某单元
            V(full)
            V(mutex)
        end
    remove(data):
        begin
            P(full)
            P(mutex)
            取缓冲区中某单元数据
            V(avail)
            V(mutex)
        end
```

在上例中，由于一个过程中包含几个公用信号量和私用信号量，因此，P、V 原语的操作次序要非常小心。一般说来，由于 V 原语是释放资源的，所以可以以任意次序出现。但 P 原语则不然，如果次序混乱，将会造成进程之间的死锁。关于死锁，将在 3.8 节中介绍。

## 3.7 进程通信

本节介绍进程间互相传递信息的方法和原理。通信(communication)意味着在进程间传送数据。操作系统可以被看作是由各种进程组成的，例如用户进程、计算进程和打印进程等。这些进程都具有各自的独立功能，且大多数被外部需要而启动执行。一般来说，进程间的通信根据通信内容可以划分为两种，即控制信息的传送与大批量数据传送。有时，也把进程间控制信息的交换称为低级通信，而把进程间大批量数据的交换称为高级通信。3.5 节和 3.6 节中介绍的进程间互斥或同步也是使用锁或信号量进行通信来实现的。低级通信一般只传送一个或几个字节的信息，以达到控制进程执行速度的作用；高级通信则要传送大量数据，其目的不是为了控制进程的执行速度，而是为了交换信息。

### 3.7.1 进程的通信方式

在单机系统中，进程间通信可分为 4 种形式：

(1) 主从式；

(2) 会话式；

(3) 消息或邮箱机制；

(4) 共享存储区方式。

主从式(master/servant system)通信系统的主要特点如下：

(1) 主进程可自由地使用从进程的资源或数据。

(2) 从进程的动作受主进程的控制。

(3) 主进程和从进程的关系是固定的。

主从式通信系统的典型例子是终端控制进程和终端进程。

会话系统(dialogue system)中，通信进程双方可分别称为使用进程和服务进程。其中，

使用进程调用服务进程提供的服务。它们具有如下特点：

（1）使用进程在使用服务进程所提供的服务之前，必须得到服务进程的许可。

（2）服务进程根据使用进程的要求提供服务，但对所提供服务的控制由服务进程自身完成。

（3）使用进程和服务进程在进行通信时有固定连接关系。

用户进程与磁盘管理进程之间的通信是会话系统的一个例子。各用户进程向磁盘管理进程提出使用要求并得到许可之后，才可以使用相应的存储区。而且，由磁盘管理进程自身完成对磁盘存储区的管理和控制。另外，用户进程与磁盘管理进程之间，只有在用户进程要求使用磁盘存储区时才有通信关系。

消息或邮箱机制则无论接收进程是否已准备好接收消息，发送进程都将把所要发送的消息送入缓冲区或邮箱。这里，消息（message）是用来区别于命令（command）或指令（instruction）等用语的。除了表示所交换的数据传递大量信息之外，消息还具有两个互相通信的进程地位平等的意思。消息的一般形式由 4 个部分组成，即发送进程名、接收进程名、数据和有关数据的操作（见图 3.16）。

消息或邮箱机制的特点如下：

（1）只要存在空缓冲区或邮箱，发送进程就可以发送消息。

（2）与会话系统不同，发送进程和接收进程之间无直接连接关系，接收进程可能在收到某个发送进程发来的消息之后，又转去接收另一个发送进程发来的消息。

（3）发送进程和接收进程之间存在缓冲区或邮箱（见图 3.17）用来存放被传送消息。




图 3.16　消息的组成

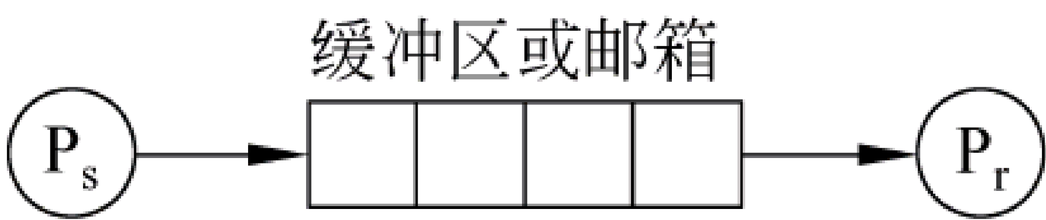



图 3.17　缓冲区或邮箱通信结构

与前面 3 种方式不同，共享存储区方式不要求数据移动。两个需要互相交换信息的进程通过对同一共享数据区（shared memory）的操作来达到互相通信的目的。这个共享数据区是每个互相通信进程的一个组成部分。

以上几种通信方式都可用于大量数据传送，而且，由于其通信方式不同，需要使用不同的控制方式来达到通信进程之间同步或互斥的目的。

下面，首先介绍进程通信中较为常用的消息与邮箱机制，然后再介绍几个实际例子。

### 3.7.2　消息缓冲机制

Hansen 在 1973 年首先提出了用消息缓冲作为进程通信的一种基本方式。发送进程和接收进程采用消息缓冲机制进行数据传送时，发送进程在发送消息前，先在自己的内存空间设置一个发送区，把欲发送的消息填入其中，然后再用发送过程将其发送出去。接收进程则在接收消息之前，在自己的内存空间内设置相应的接收区，然后用接收过程接收消息。由于消息缓冲机制中所使用的缓冲区为公用缓冲区，使用消息缓冲机制传送数据时，两个通信进程必须满足如下条件：

（1）在发送进程把消息写入缓冲区和把缓冲区挂入消息队列时，应禁止其他进程对该缓冲区消息队列的访问。否则，将引起消息队列的混乱。同理，当接收进程正从消息队列中

取消息缓冲时,也应禁止其他进程对该队列的访问。

(2) 当缓冲区中无消息存在时,接收进程不能接收到任何消息。

至于发送进程是否可以发送消息,则由发送进程是否申请到缓冲区决定。

设公用信号量 mutex 为控制对缓冲区访问的互斥信号量,其初值为 1。设 SM 为接收进程的私用信号量,表示等待接收的消息个数,其初值为 0。设发送进程调用过程 Send(m) 将消息 m 送往缓冲区,接收进程调用过程 Receive(m)将消息 m 从缓冲区读往自己的数据区,则 Send(m)和 Receive(n)可分别描述为

```
Send(m):
    begin
        向系统申请一个消息缓冲区
        P(mutex)
        将发送区消息 m 送入新申请的消息缓冲区
        把消息缓冲区挂入接收进程的消息队列
        V(mutex)
        V(SM)
    end
Receive(n):
    begin
        P(SM)
        P(mutex)
        摘下消息队列中的消息 n
        将消息 n 从缓冲区复制到接收区
        释放缓冲区
        V(mutex)
    end
```

一般来说,尽管系统中可利用的缓冲区总数是已知的,但由于消息队列是按接收进程排列,因而,在同一时间内,系统中存在着多个消息队列;且这些队列的长度是不固定的。因此,发送进程无法在 Send 过程用 P 操作判断信号量 SM。

### 3.7.3 邮箱通信

邮箱通信就是由发送进程申请建立一个与接收进程链接的邮箱。发送进程把消息送往邮箱,接收进程从邮箱中取出消息,从而完成进程间的信息交换。设置邮箱的最大好处就是发送进程和接收进程之间没有处理时间上的限制。一个邮箱可考虑成发送进程与接收进程之间的大小固定的私有数据结构,它不像缓冲区那样被系统内所有进程共享。邮箱由邮箱头和邮箱体组成。其中邮箱头描述邮箱名称、邮箱大小、邮箱方向以及拥有该邮箱的进程名等。邮箱体主要用来存放消息(见图 3.18)。

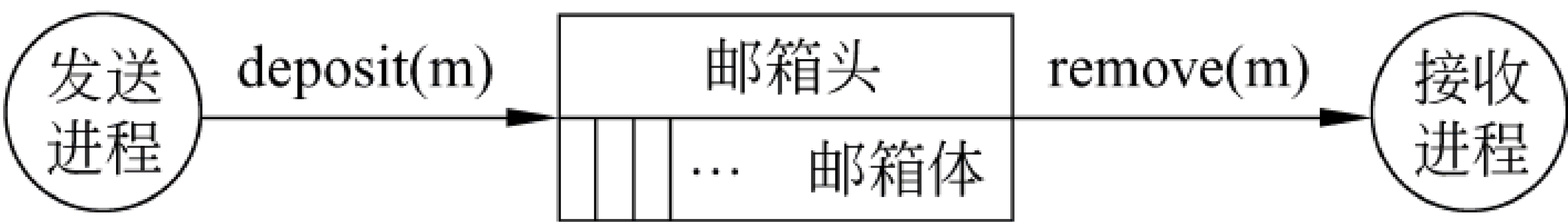



图 3.18 邮箱通信结构

邮箱通信的特点已在前面介绍过，对于只有一个发送进程和一个接收进程使用的邮箱，则进程间通信应满足如下条件：

(1) 发送进程发送消息时，邮箱中至少要有一个空格能存放该消息。

(2) 接收进程接收消息时，邮箱中至少要有一个消息存在。

设发送进程调用过程 deposit(m)将消息发送到邮箱，接收进程调用过程 remove(m)将消息 m 从邮箱中取出。另外，为了记录邮箱中空格个数和消息个数，信号量 fromnum 为发送进程的私用信号量，信号量 mesnum 为接收进程的私用信号量。fromnum 的初值为信箱的空格数 $n$，mesnum 的初值为 0。则 deposit(m)和 remove(m)可描述如下：

```
deposit(m):
    begin local x
        P(fromnum)
        选择空格 x
        将消息 m 放入空格 x 中
        置格 x 的标志为满
        V(mesnum)
    end
remove(m):
    begin local x
        P(mesnum)
        选择满格 x
        把满格 x 中的消息取出放入 m 中
        置格 x 标志为空
        V(fromnum)
    end
```

显然，调用过程 deposit(m)的进程与调用过程 remove(m)的进程之间存在着同步制约关系而不是互斥制约关系。

另外，在许多时候，存在着多个发送进程和多个接收进程共享邮箱的情况。这时需要对过程 deposit(m)和 remove(m)作相应的改动。

### 3.7.4 进程通信的实例——和控制台的通信

通用计算机中，除了用户终端之外，还有一台由系统操作员控制的控制台终端。各用户进程可将消息送到控制台进程，操作员在读到这些消息后做出相应的处理。

设控制台终端由键盘和显示器组成，终端和主机之间按全双工模式发送和接收数据，即键盘和数据显示彼此独立。设键盘控制进程和显示控制进程分别为 KCP 和 DCP，用户进程和控制台终端的通信由会话控制进程 CCP 控制完成。其中控制台键盘的输入放入缓冲队列 inbuf 中，CCP 可从 inbuf 中取出消息从而得到来自控制台的指示。而 CCP 所提出的问题则以消息形式放入控制台的输出缓冲队列 outbuf 中，DCP 从 outbuf 中取出消息送至显示器，以供操作员判断。其通信过程如图 3.19 所示。

下面，先描述 KCP 与键盘、DCP 与显示器之间的通信动作，然后描述 CCP 与 KCP 及 DCP 的接口，紧接着再给出 CCP 与用户进程的接口，最后再综合描述 CCP 的动作。

DCP: Display Control Process
KCP: Keyboard Control Process
CCP: Convesation Control Process
$P_1, P_2, \cdots, P_n$: user processes

图 3.19 和控制台通信示例

**1. KCP 和 DCP 的动作**

以上通信过程是一个严格的提问/回答方式的通信例子。先描述 KCP 和 DCP 的动作(下面的描述中省略了进程的 PCB 部分)。

首先,当操作员按键时,KCP 将对应的数据从键盘送入输入缓冲 inbuf 中,同时,也将输入数据送 echobuf 在显示器上显示。显然,KCP 和 CCP 等之间的通信满足消息机制的条件。另外,除了 KCP 和 CCP 的通信之外,KCP 实际上还在和键盘发生通信。因此,在描述 KCP 和 CCP 时,还必须考虑 KCP 和键盘的通信。

KCP 可描述如下:

键盘控制进程 KCP:

```
设 T-Ready 和 T-Busy 分别为键盘动作 KP 和键盘控制进程 KCP 的私用信号量,其初值为 0 和 1
   初始化 {清除所有 inbuf 和 echobuf}
   begin local x
       P(T-Ready)
       从键盘数据传输缓冲 x 中取出字符 m,记为 x.m
       Send(x.m)
       将 x.m 送入 echobuf
       V(T-Busy)
   end
```

这里,假设键盘也可用进程描述如下:

键盘动作 KP:

```
repeat  local x
        P(T-Busy)
        把输入字符放入数据传输缓冲 x
        V(T-Ready)
until 终端关闭
```

同样,可以写出 DCP 和显示器动作 DP 的通信动作:

显示器控制进程 DCP:

```
设 D-Ready 和 D-Busy 分别为 DP 和 DCP 的私用信号量,其初值为 0 和 1
```

```
初始化{清除输出缓冲 outbuf,echo 模式置 false}
begin
    if outbuf 满
    then
        receive(k)/* CCP k*/
        P(D-Busy)
        把 k 送入显示器数据缓冲区
        V(D-Ready)
    else
        echo 模式置 true
        echobuf 中字符置入显示器数据缓冲区
    fi
```

显示器动作 DP：

```
repeat
    if  echo 模式
    then
        打印显示器数据缓冲区中的字符
    else
        P(D-Ready)
        打印显示器数据缓冲区中的消息
        V(D-Busy)
    fi
until 显示器关机
```

这里，假定一个消息的长度总是小于 outbuf 的长度。

**2. CCP 和 KCP 及 DCP 的接口**

上面已经描述了 KCP 与 KP、DCP 与 DP 的同步动作部分。那么，KCP、DCP 与 CCP 之间又是如何动作的呢？现在来看 CCP 怎样从 inbuf 中读出消息和怎样把消息写入 outbuf。这里，仍然假定一个消息的长度小于 outbuf 和 inbuf 的长度。

设过程 Read(x)把 inbuf 中的所有字符读到用户进程数据区 x 处，过程 Write(y)把用户进程 y 处的消息写到 outbuf 中，则 Read(x)和 Write(y)可分别描述如下：

```
Read(x):
    begin
        P (inbuf-full)
        Copy(inbuf into x)
        V (inbuf-empty)
    end
Write(y):
    begin
        P (outbuf-empty)
        Copy(outbuf from y)
        V(outbuf-full)
    end
```

这里，inbuf-full 和 inbuf-empty 分别是 CCP 和 KCP 的私用信号量，在过程 Send(m)和 Read(x)中使用，其初值分别为 0 和 1。由于在 KCP 中，Send(m)被用来将字符一个个地送

入 inbuf 中,Send(m)过程必须要做一定的改动,也就是要加入缓冲计数功能和把 inbuf 的长度看作是固定的。关于 Send(m)的修改请读者自己完成。另外,outbuf-full 和 outbuf-empty 则是 DCP 和 CCP 的私用信号量,在过程 receive(k)和 Write(y)中使用,其初值分别为 0 和 1。由于 outbuf 的长度是固定的,所以 receive(k)过程也应作相应的修改。receive(k)的修改也请读者自己完成。

**3. CCP 与用户进程的接口**

除了和 KCP 及 DCP 的通信之外,CCP 还要从各用户进程那里得到提问和向提问的用户进程转达从控制台来的指示。因此,CCP 和用户进程之间也存在着通信关系。

设各用户进程向 CCP 发出的提问用消息组成队列 RQ。各用户进程把消息送入 RQ 时必须互斥操作,否则将引起 RQ 队列混乱。因此,设互斥用信号量 rq,初值为 1。另外,CCP 只有在用户进程提问之后才负责向控制台转发提问和向用户进程转达控制台的指示。因此,还必须为 CCP 设置一个私用信号量 question 以计算用户进程所提出的问题数目。信号量 question 的初值为 0。另外,由于各用户进程在 CCP 发出回答消息之后,不一定马上就能处理 CCP 所发出的回答消息,因而,需设置相应的消息接收队列 $SQ_i$。$SQ_i$ 和 RQ 的关系如图 3.20 所示。

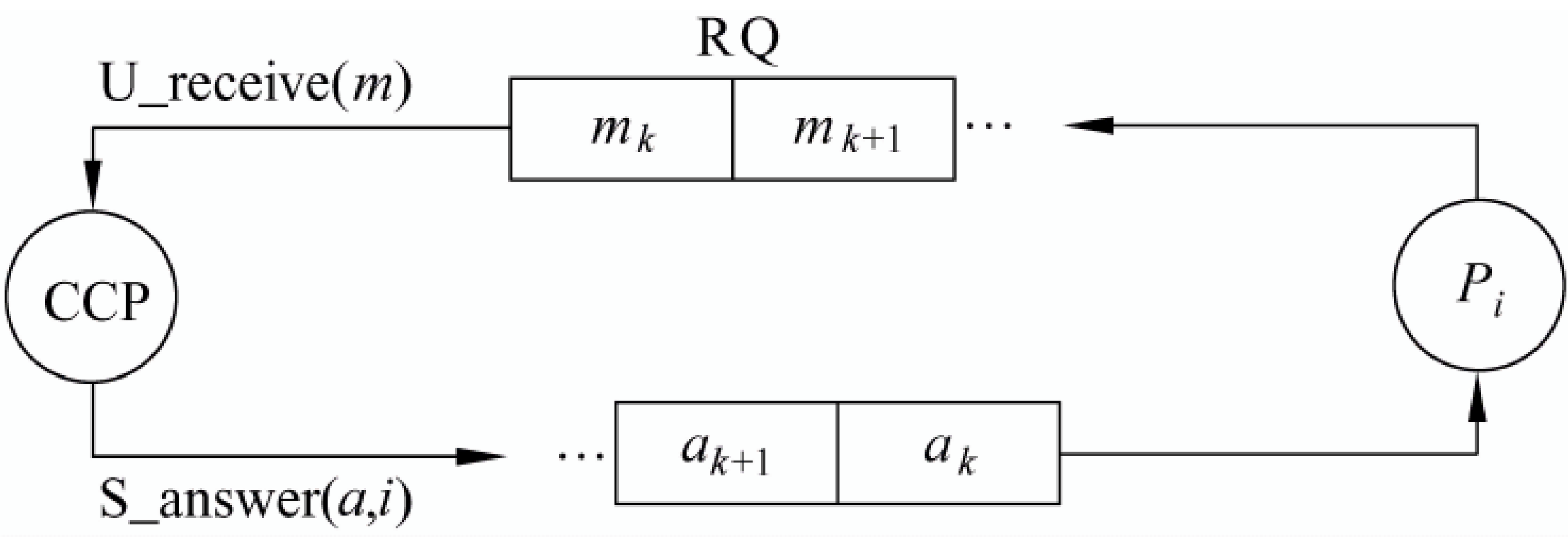



图 3.20 CCP 和用户进程接口

与对 RQ 队列的操作相同,对 $SQ_i$ 队列的操作也必须是互斥的,因而,设互斥信号量 $sq_i$,其初值为 1。除此之外,为了保证 CCP 和各用户进程之间获得回答的同步,还需设一个私用信号量 $answer_i$ 以计算 $SQ_i$ 中的消息个数。

CCP 和用户进程的接口可描述如下:

```
U-receive(m):
    begin
        P (question)
        when rq do 从 RQ 中取出 m od
        return (m)
    end
```

这里,采用了互斥变量 rq 作为临界区名。U-receive(m)描述从 RQ 取出一个消息的过程。

```
S-answer(a,i):
    begin
        when sqi do 把 a 插入 SQi 中 od
        V (answeri)
    end
```

**4. CCP 的动作**

作为实现用户进程与控制台通信的方法之一,利用上面所述的各种接口,可以描述实现

严格的顺序会话 CCP 进程。

会话控制进程 CCP：

```
local k,m,x
    repeat
        U-receive(m)
    将消息 m 的进程标号置入 k 中
    将消息 m 解码变换到 x
    Write(x)
    Read(x)
    将 x 编码到 m
    S-answer(m,k)
until CCP 结束
```

### 3.7.5 进程通信的实例——管道

**1. 管道(pipe)**

进程通信的实例之一是 UNIX 系统的管道通信。UNIX 系统从 System Ⅴ开始，提供有名管道和无名管道两种数据通信方式，这里介绍无名管道。

无名管道为建立管道的进程及其子孙提供一条以比特流方式传送消息的通信管道。该管道在逻辑上被看作管道文件，在物理上则由文件系统的高速缓冲区构成，而很少启动外设。发送进程利用文件系统的系统调用 write(fd[1],buf,size)把 buf 中的长度为 size 个字符的消息送入管道入口 fd[1]，接收进程则使用系统调用 read(fd[0],buf,size)从管道出口 fd[0]读出 size 个字符的消息置入 buf 中。这里，管道按 FIFO 方式传送消息，且只能单向传送消息(见图 3.21)。

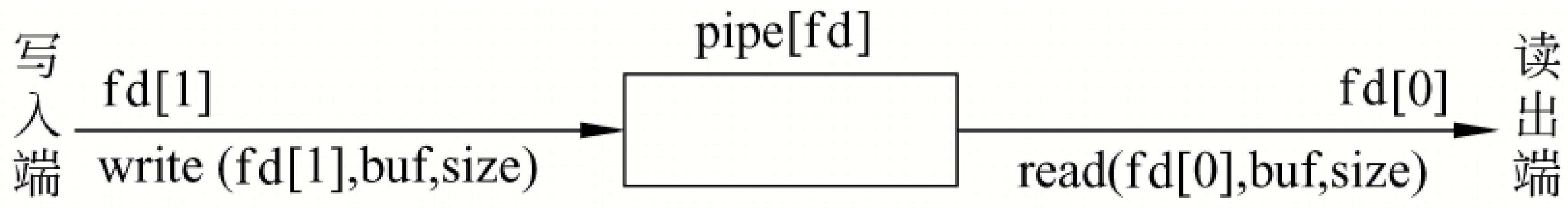



图 3.21 管道通信

利用 UNIX 提供的系统调用 pipe 可建立一条同步通信管道。其格式为

```
pipe(fd)
int fd[2];
```

这里，fd[1]为写入端，fd[0]为读出端。

**2. 示例**

**例 1**：用 C 语言编写一个程序，建立一个管道，同时父进程生成一个子进程，子进程向管道中写入一个字符串，父进程从管道中读出该字符串。

**解**：程序如下：

```
#include <stdio.h>
main()
{
    int x,fd[2];
```

```
    char buf[30],s[30];
    pipe(fd);                                /* 创建管道 */
    while((x=fork())==-1);                   /* 创建子进程失败时,循环 */
    if(x==0)
    {
        sprintf(buf,"This is an example\n");
        write(fd[1],buf,30);                 /* 把 buf 中的字符写入管道 */
        exit(0);
    }
    else                                     /* 父进程返回 */
    {
        wait(0);
        read(fd[0],s,30);                    /* 父进程读管道中的字符 */
        printf("%s",s);
    }
}
```

**例 2**：编写一个程序,建立一个管道。同时,父进程生成子进程 $P_1$ 和 $P_2$,这两个子进程分别向管道中写入各自的字符串,父进程读出它们(如图 3.22 所示)。

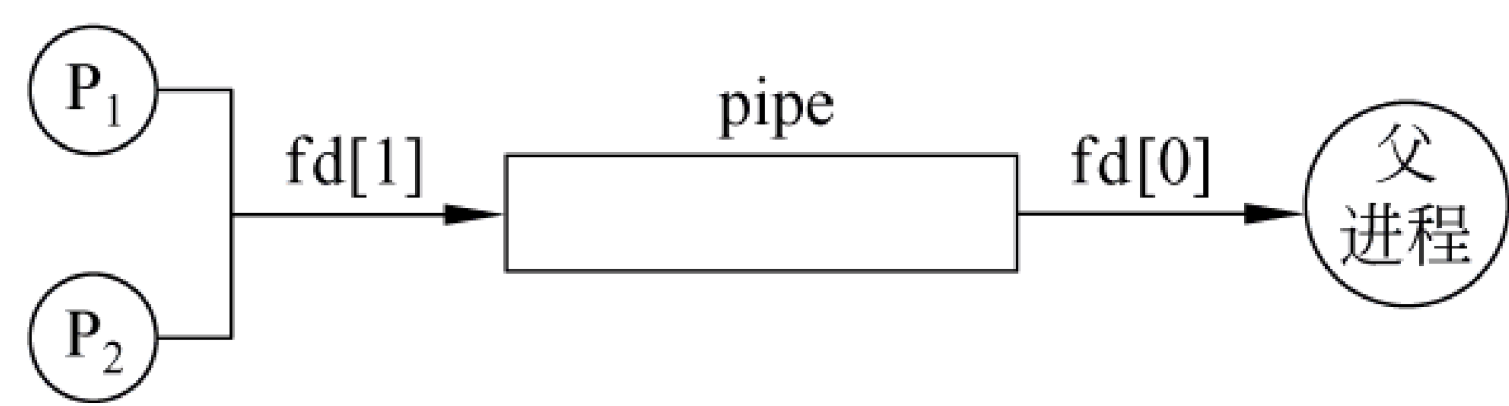



图 3.22　父进程和子进程 $P_1$、$P_2$ 通信

**解**：程序框图如图 3.23 所示,源程序如下：

```
#include <stdio.h>
main()
{
    int  i,r,p1,p2,fd[2];
    char buf[50],s[50];
    pipe(fd);                                /* 父进程建立管道 */
    while((p1=fork())==-1);                  /* 创建子进程 P1,失败时循环 */
    if(p1==0)                                /* 由子进程 P1 返回,执行子进程 P1 */
    {
        lockf(fd[1],1,0);                    /* 加锁锁定写入端 */
        sprintf(buf,"child process P1 is sending messages!\n");
        printf("child process P1!\n");
        write(fd[1],buf,50);                 /* 把 buf 中的 50 个字符写入管道 */
        sleep(5);                            /* 睡眠 5 秒,让父进程读 */
        lockf(fd[1],0,0);                    /* 释放管道写入端 */
        exit(0);                             /* 关闭 P1 */
    }
    else                                     /* 从父进程返回,执行父进程 */
    {
        while((p2=fork())==-1);              /* 创建子进程 P2,失败时循环 */
```

```
        if(p2==0)                          /* 从子进程 P2 返回,执行 P2 */
        {
            lockf(fd[1],1,0);              /* 锁定写入端 */
            sprintf(buf,"child process P2 is sending messages!\n");
            printf("child process P2 !\n");
            write(fd[1],buf,50);           /* 把 buf 中的字符写入管道 */
            sleep(5);                      /* 睡眠等待 */
            lockf(fd[1],0,0);              /* 释放管道写入端 */
            exit(0);                       /* 关闭 P2 */
        }
        wait(0);
        if(r=read(fd[0],s,50)==-1)
            printf("can't read pipe\n");
        else printf("%s\n",s);
        wait(0);
        if(r=read(fd[0],s,50)==-1)
            printf("can't read pipe\n");
        else printf("%s\n",s);
        exit(0);
    }
}
```

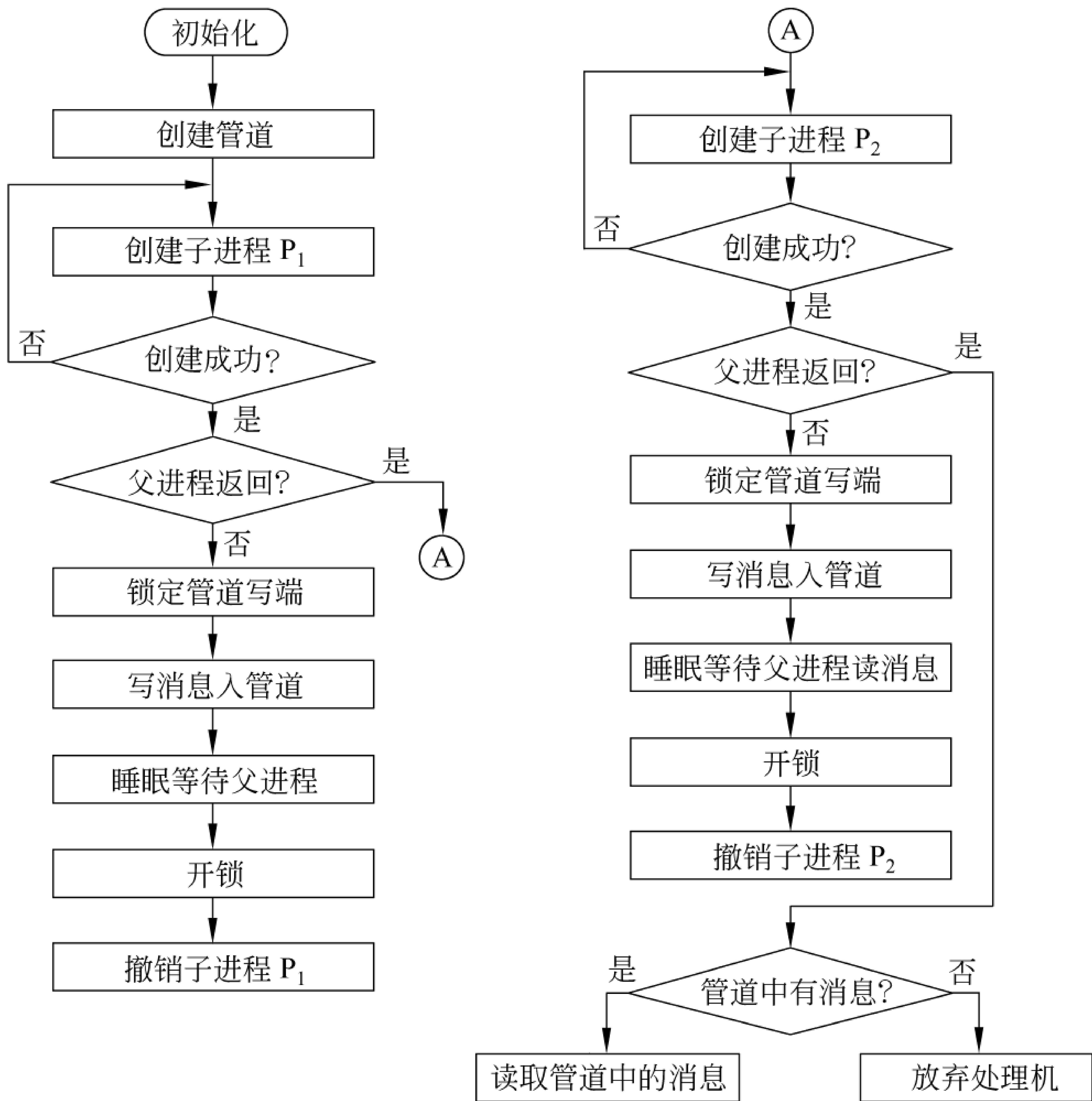



图 3.23　例 2 程序流图

其中，lockf 为保证进程互斥使用管道的系统调用，sleep 为保证当前进程睡眠以转让处理机的系统调用。

## 3.8 死锁问题

### 3.8.1 死锁的概念

**1. 死锁的定义**

各进程在使用系统资源时，应注意系统产生死锁问题。所谓死锁，是指各并发进程互相等待对方所拥有的资源，且这些并发进程在得到对方的资源之前不会释放自己所拥有的资源。从而造成大家都想得到资源而又都得不到资源，各并发进程不能继续向前推进的状态。图 3.24 是两个进程发生死锁时的例子。

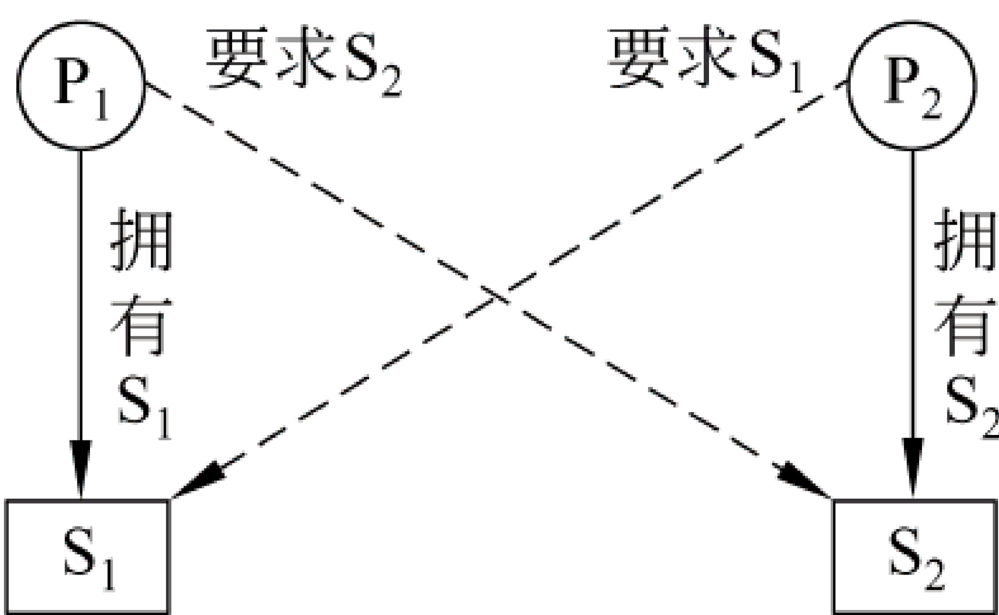



图 3.24 死锁的概念

下面以生产者/消费者问题为例来进一步看看死锁的概念。设生产者进程已获得对缓冲区队列的操作权，生产者进程进一步要求对缓冲区内的某一空缓冲区进行置入消息操作；然而，设此时缓冲队列内所有缓冲区都是满的，即只有消费者进程才能对它们进行取消息操作，因此，生产者进程进入等待状态。反过来，消费者进程拥有对各缓冲区操作的操作权，为了对各缓冲区进行操作，它又要申请对缓冲队列操作的操作权；由于对缓冲队列的操作权被生产者进程掌握，且生产者进程不会自动释放它，从而消费者进程也只能进入等待状态而陷入死锁。

一般地，可以把死锁描述为：有并发进程 $P_1, P_2, \cdots, P_n$，它们共享资源 $R_1, R_2, \cdots, R_m$（$n>0, m>0, n\geqslant m$）。其中，每个 $P_i$（$1\leqslant i\leqslant n$）拥有资源 $R_j$（$1\leqslant j\leqslant m$），直到不再有剩余资源。同时，各 $P_i$ 又在不释放 $R_j$ 的前提下要求得到 $R_k$（$k\neq j, 1\leqslant k\leqslant m$），从而造成资源的互相占有和互相等待。在没有外力驱动的情况下，该组并发进程停止往前推进，陷入永久等待状态。

**2. 死锁的起因**

死锁的起因是并发进程的资源竞争。产生死锁的根本原因在于系统提供的资源个数少于并发进程所要求的该类资源数。显然，由于资源的有限性，不可能为所有要求资源的进程无限制地提供资源。但是，可以采用适当的资源分配算法，以达到消除死锁的目的。为此，先看看产生死锁的必要条件。

**3. 产生死锁的必要条件**

从死锁的概念，可以得到产生死锁的必要条件如下：

（1）互斥条件。并发进程所要求和占有的资源是不能同时被两个以上进程使用或操作的，进程对它所需要的资源进行排他性控制。

（2）不剥夺条件。进程所获得的资源在未使用完毕之前，不能被其他进程强行剥夺，而只能由获得该资源的进程自己释放。

（3）部分分配。进程每次申请它所需要的一部分资源，在等待新资源的同时，继续占用已分配到的资源。

（4）环路条件。存在一种进程循环链，链中每一个进程已获得的资源同时被下一个进程所请求。

显然，只要使上述 4 个必要条件中的某一个不满足，死锁就可以消除。

### 3.8.2 死锁的消除方法

解决死锁的方法一般可分为预防、避免、检测与恢复 3 种。预防是采用某种策略，限制并发进程对资源的请求，从而使得死锁的必要条件在系统执行的任何时间都不满足。避免是指系统在分配资源时，根据资源的使用情况提前做出预测，从而避免死锁的发生。死锁检测与恢复是指系统设有专门的机构，当死锁发生时，该机构能够检测到死锁发生的位置和原因，并能通过外力破坏死锁发生的必要条件，从而使得并发进程从死锁状态中恢复出来。

一般而言，由于操作系统的并行与共享以及随机性等特点，通过预防和避免的手段达到消除死锁的目的是一件十分困难的事。这需要较大的系统开销，甚至不能充分利用资源。死锁的检测和恢复则与此相反，不必花费多少执行时间就能发现死锁和从死锁中恢复出来。因此，在实际操作系统中大都使用检测与恢复法消除死锁。

**1. 死锁预防**

怎样预防死锁呢？一种方法是打破资源的互斥和不可剥夺这两个条件，例如允许进程同时访问某些资源等。然而，这种方法不能解决访问那些不允许被同时访问的资源时所带来的死锁问题，比如打印机等。另一种方法则是打破资源的部分分配这个死锁产生的必要条件。即预先分配各并发进程所需要的全部资源。如某个进程的资源得不到满足时，则安排一定的等待次序让其他进程释放资源。但是，这种方法也有如下缺点：

（1）在许多情况下，一个进程在执行之前不可能提出它所需要的全部资源。

（2）无论所需资源何时用到，一个进程只有在所有要求资源都得到满足之后才开始执行。

（3）对于那些不经常使用的资源，进程在生存过程期间一直占用它们是一种极大的浪费。

（4）降低了进程的并发性。

另外一种死锁的预防方法是打破死锁的环路条件。即把资源分类按顺序排列，使进程在申请、保持资源时不形成环路。如有 $m$ 种资源，则列出 $R_1<R_2<\cdots<R_m$。若进程 $P_i$ 保持了资源 $R_i$，则它只能申请比 $R_i$ 级别更高的资源 $R_j(R_i<R_j)$。释放资源时必须是 $R_j$ 先于 $R_i$ 被释放，从而避免环路的产生。这种方法的缺点是限制了进程对资源的请求，而且对资源的分类编序也耗去一定的系统开销。

**2. 死锁避免**

死锁避免可被称为动态预防，因为系统采用动态分配资源，在分配过程中预测出死锁发生的可能性并加以避免的方法。

死锁避免的一种基本模式是把进程分为多个步，其中每个步所使用的资源是固定的，且在一个步内，进程所保持的资源数不变。即进程的资源请求、使用与释放要依靠不同的步完成。

设并发进程 $P_1,P_2,\cdots,P_n(n\geqslant 1)$ 共享不同类型的资源 $R_1,R_2,\cdots,R_m(m\geqslant 1)$，每一 $R_i$ 有固定的单元数目 $C_i(1\leqslant i\leqslant n)$。系统按一定的资源分配算法给各进程分配资源。

可用一个向量矩阵<$\boldsymbol{W},\boldsymbol{A},\boldsymbol{B},\boldsymbol{F}$>来描述各进程的资源请求和获得系统空闲资源的状况。其中,$\boldsymbol{W}=(\boldsymbol{W}_1,\boldsymbol{W}_2,\cdots,\boldsymbol{W}_n)$是一个 $n\times m$ 维矩阵,$n$ 表示并发进程数目,$m$ 表示资源类型数目,$\boldsymbol{W}_{ij}=w_i(j)$是进程 $P_i$ 在某一执行步时为完成任务请求追加的资源 $R_j$ 的单元数目。$\boldsymbol{W}_i$ 被称为进程 $P_i$ 的请求向量。$\boldsymbol{A}=(\boldsymbol{A}_1,\boldsymbol{A}_2,\cdots,\boldsymbol{A}_n)$是 $n\times m$ 维分配矩阵,$\boldsymbol{A}_{ij}=a_i(j)$是系统分配给进程 $P_i$ 的资源 $R_j$ 的单元数,$\boldsymbol{A}_i$ 被称为分配向量。$\boldsymbol{B}=(\boldsymbol{B}_1,\boldsymbol{B}_2,\cdots,\boldsymbol{B}_n)$是 $n\times m$ 维释放矩阵,$\boldsymbol{B}_{ij}=b_i(j)$是进程 $P_i$ 释放的资源 $R_j$ 的单元数。$\boldsymbol{F}=(f_1,f_2,\cdots,f_m)$是空闲资源向量。

设 $\boldsymbol{C}=(c_1,c_2,\cdots,c_m)$为系统能力向量,则有

$$f_j = c_j - \sum_{i=1}^{n} a_i(j)$$

即 $R_j$ 类资源的空闲单元数是总单元数减去已分配给各进程的单元数。

设进程 $P_i$ 可被分为 $k$ 个步 $P_i(1),P_i(2),\cdots,P_i(k)$,其中 $P_i(1)$为初始步且 $P_i(k)$为终止步,而且,在每一步中进程保持资源和请求资源,在每一步执行结束后进程释放资源和系统分配资源。若对于 $P_i(1),P_i(2),\cdots,P_i(k)$有

$$w_i(1) \leqslant F$$

且

$$w_i(r) \leqslant \boldsymbol{F} + \sum_{j=1}^{m} b_i(j)$$

成立,则进程 P 在结束序列中。如果所有并发进程都在结束序列中,则系统是安全的(无死锁)。也就是说,进程的请求向量不能大于空闲资源和该进程准备释放的资源。这里 $\boldsymbol{W}_i(r)$表示进程 $P_i$ 第 $r$ 步时的请求向量。

显然,死锁回避需要占去系统较大的开销。

**3. 死锁的检测和恢复**

当进程进行资源请求时,死锁检测算法检查并发进程组是否构成资源的请求和保持环路。有限状态转移图和 PetriNet 等技术都可用来有效地判断死锁发生。死锁的恢复办法较多,最简单的办法是终止各锁住进程,或按一定的顺序中止进程序列,直至已释放到有足够的资源来完成剩下的进程时为止。另外,也可以从被锁住进程强迫剥夺资源以解除死锁。

## 3.9 线程的概念

### 3.9.1 为什么要引入线程

进程是为了提高 CPU 的执行效率,减少因为程序等待带来的 CPU 空转以及其他计算机软硬件资源的浪费而提出来的。进程是为了完成用户任务所需要的程序的一次执行过程以及为其分配资源的一个基本单位。以进程为单位来分配资源时,为了便于处理机执行,系统要定义如何对进程进行识别和操作的物理实体。在学习进程管理时,我们已经知道,这个被系统识别和操作的物理实体就是进程控制块(PCB)。PCB 负责记录进程的描述信息、有关控制信息、各种资源的管理信息和所对应进程的 CPU 现场保护信息等(执行中的进程除外)。

由进程管理部分知道,在网络或多用户环境下,许多用户的应用任务都是并发进行的,

而且，它们往往只有较多的软硬件资源。在引入了进程的概念之后，这些用户的应用任务都以进程的方式管理、执行和完成。

然而，在许多情况下，用户所要完成的任务具有许多相似的性质。例如，一个 Web 服务器可以同时接收来自不同用户的网页访问请求。显然，服务器处理这些网页请求都是并发进行的，否则，将会造成用户等待时间过长和响应时间降低。

如果在服务器中用进程的办法来处理来自不同用户的网页访问请求的话，我们可以用创建父进程和多个子进程的方式来进行处理。

由本章前面的内容可知，创建一个进程要花费较大的系统开销和占用较多的资源。例如，至少要消费一个 PCB 结构。如果这个进程创建的子进程过多(一般来说，随着访问服务器的用户数增加，子进程数量将增加)，则使用的进程 PCB 结构和其他系统资源越多。

显然，对于具有不确定用户和随机访问的 Web 服务器而言，用进程来管理用户访问请求的方法会较大地限制开发访问服务器的用户数。

另一方面，当不同的用户请求访问 Web 服务器时，不同的用户子进程将被用来完成访问请求处理。这些不同的用户子进程的执行涉及进程的上下文切换。进程上下文切换是一个复杂的过程。它涉及对当前正在执行进程的状态和占用资源的保存写处理、选取新的待执行进程以及恢复待执行进程的执行状态和资源等工作。

进程的创建和切换过程越多，系统的开销就越大，从而，服务器可以处理和支持的用户访问请求就会越少。显然，如何减少进程创建和切换所带来的系统开销，是服务器操作系统的关键问题之一。

类似的例子还有许多，例如远程过程调用和远程数据库访问等。

为了减少进程切换和创建的开销，提高执行效率和节省资源，人们开始在操作系统中引入“线程”(thread)的概念。

## 3.9.2 线程的基本概念

线程是进程的一部分。

线程有时又被称为轻权进程或轻量级进程(light weight process)。

与进程相同，线程也是 CPU 调度的一个本单位。

一个没有线程的进程可以被看作是单线程的，即进程的执行过程是线状的，尽管中间会发生中断或暂停，但该进程所拥有的资源只为该线状执行过程服务。一旦发生进程上下文切换，这些资源都是要被保护起来的。单线程进程的概念如图 3.25 所示。

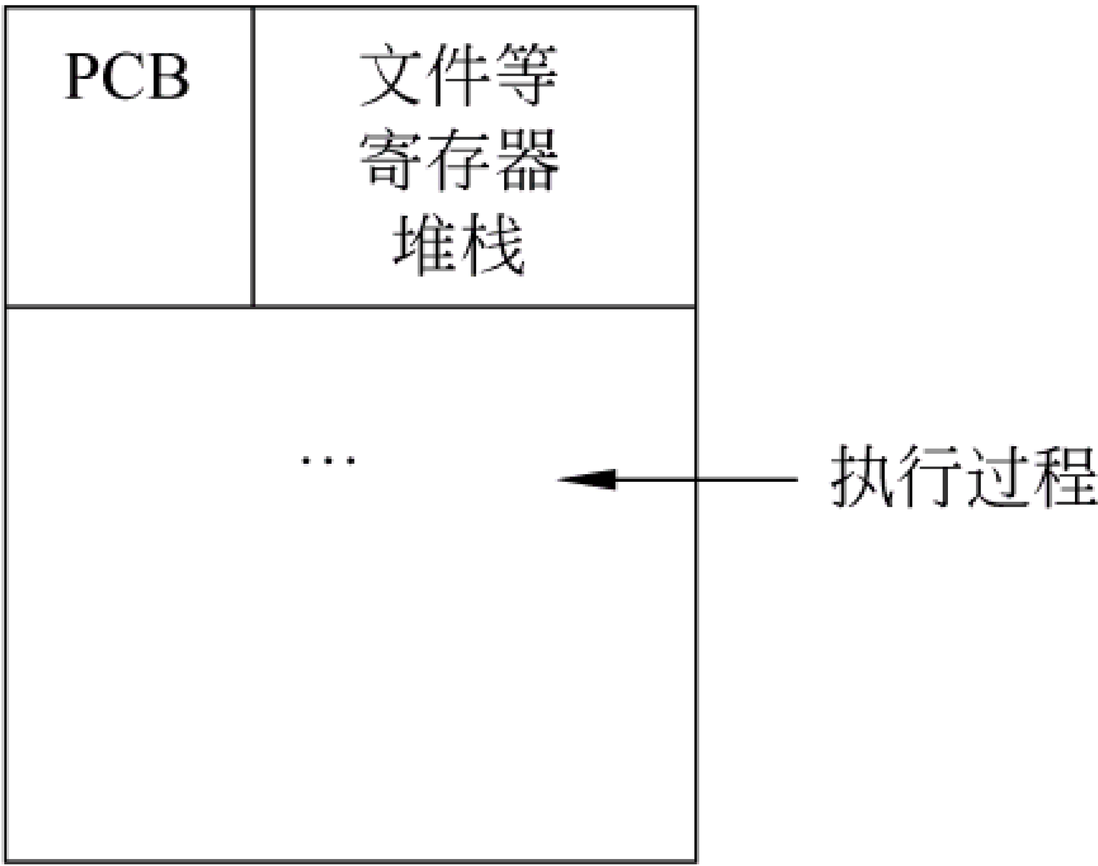


图 3.25　等效于单线程的进程执行示意图

与此相对应，如果在一个进程内拥有多个线程，则进程的执行过程不再是唯一线状的，它由多条线状执行过程组成，如图 3.26 所示。

## 3.9.3 线程与进程的区别

图 3.25 和图 3.26 已大致上给出了进程和线程的区别。从中可以看出，虽然进程和线程都是处理机调度的基本单位，但是，线程的改变只代表了 CPU 执行过程的改变，而进程

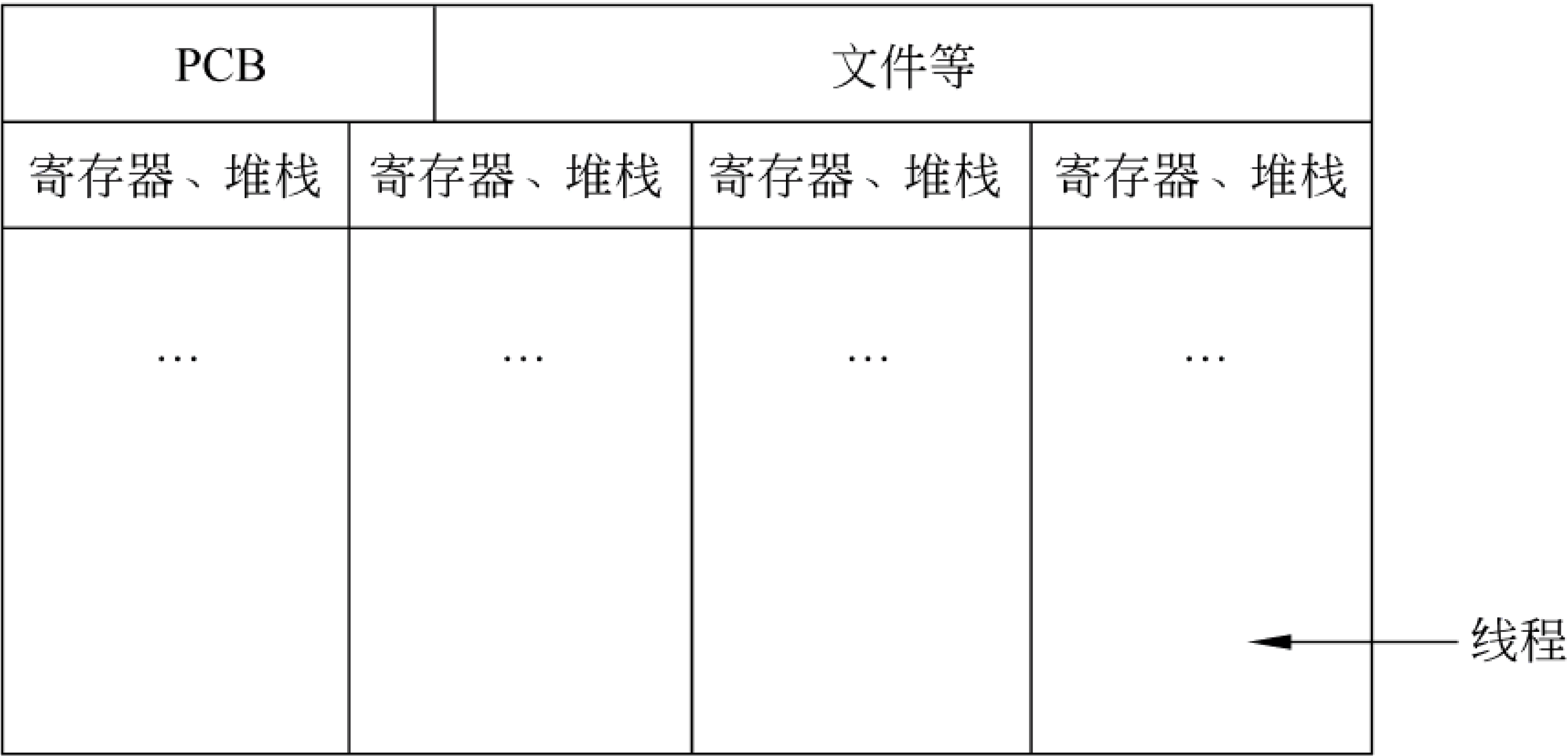


图 3.26　多线程情况下的进程执行示意图

所拥有的资源没有发生变化。或者说,除了 CPU 之外,计算机内的软硬件资源的分配与线程无关,线程只能共享它所属进程的资源。

与进程控制表和 PCB 相似,每个线程也有自己的线程控制块(TCB),而这个 TCB 中所保存的线程状态信息则要比 PCB 少得多,这些信息主要是相关指针用堆栈(系统栈和用户栈)以及寄存器中的状态数据。

进程是系统中所有资源分配时的基本单位,例如,打印机等设备都是以进程为单位进行分配的。

进程拥有一个完整的虚拟地址空间(后述)。

进程不依赖于线程而独立存在。

反之,线程是进程的一部分,它没有自己的地址空间,它和进程内的其他线程一起共享分配给该进程的所有资源。

多线程系统中进程与线程的关系如图 3.27 所示。

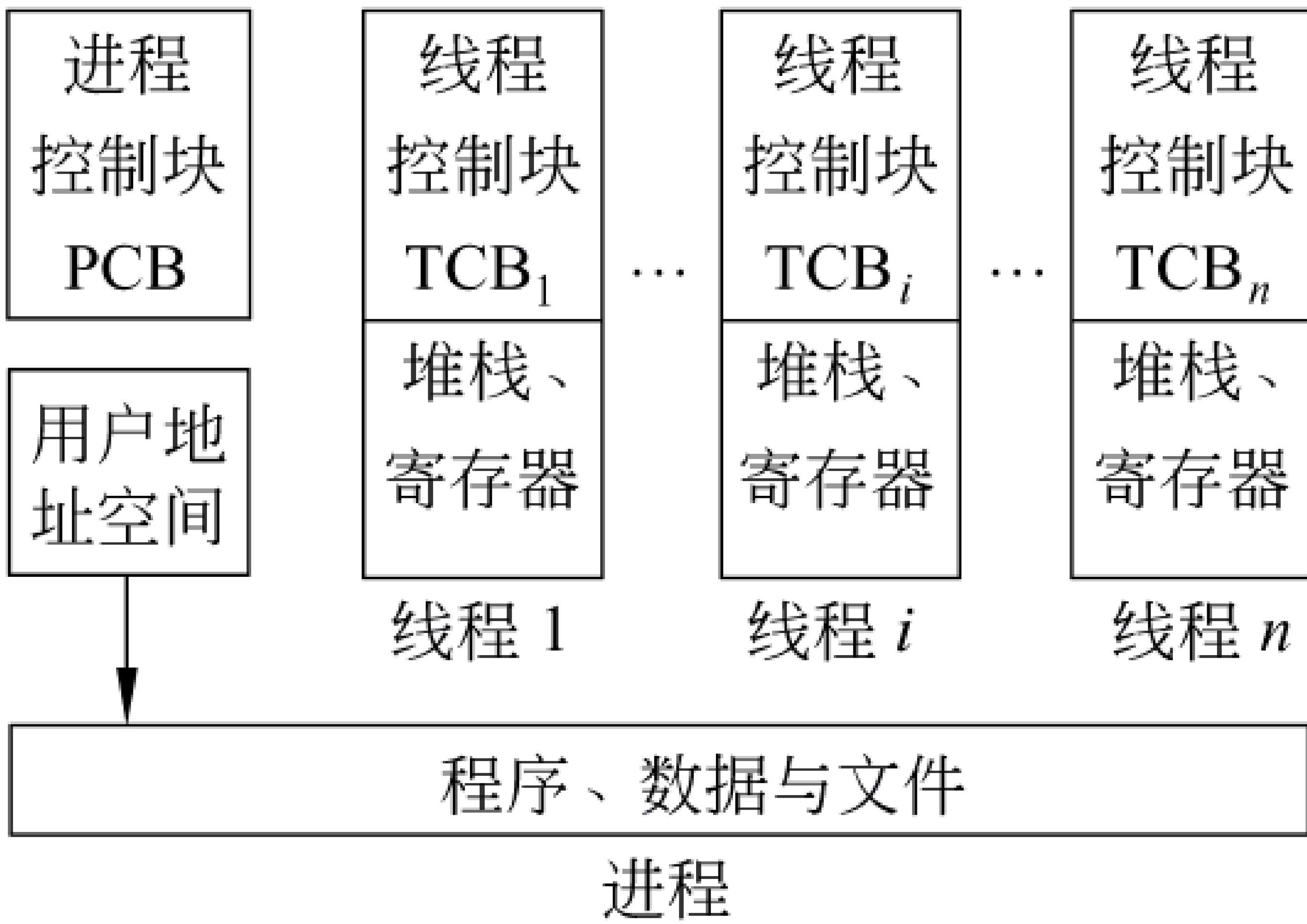


图 3.27　多线程与进程的关系

## 3.9.4　线程的适用范围

尽管线程可以提高系统的执行效率,但并不是在所有的计算机系统中线程都是适用的。事实上在那些很少做进程调度和切换的实时系统和个人数字助理系统中,由于任务的单一性,设置线程反而会占用更多的系统资源。

使用线程的最大好处是在有多个任务需要处理机处理时可以减少处理机的切换时间;而且,线程的创建和结束所需要的系统开销也比进程要小得多。由此,可以推出最适合使用线程的系统是多处理机系统、网络系统或分布式系统。在多处理机系统中,同一用户程序可以根据不同的功能划分为不同的线程,放在不同的处理机上执行。在网络或分布式系统中,服务器可对多个不同用户的请求按不同的线程进行处理,从而提高系统的处理速度和效率。

在用户程序可以按功能划分为不同的小段时,单处理机系统也可因使用线程而简化程序的结构和提高执行效率。

下面是几种典型的应用:

(1) 服务器中的文件管理或通信控制。在局域网的文件服务器中,对文件的访问要求可被服务器进程派生出的线程进行处理。由于服务器同时可能接受许多个文件访问要求,

则系统可以同时生成多个线程来进行处理。如果计算机系统是多处理机的,这些线程还可以安排到不同的处理机上执行。

(2) 前后台处理。许多用户都有过前后台处理经验,即把一个计算量较大的程序或实时性要求不高的程序安排在处理机空闲时执行。对于同一个进程中的上述程序来说,线程可被用来减少处理机切换时间和提高执行速度。例如,在表处理进程中,一个线程可被用来显示菜单和读取用户输入,而另一个线程则可用来执行用户命令和修改表格。由于用户输入命令和命令执行分别由不同的线程在前后台执行,从而提高了操作系统的效率。

(3) 异步处理。程序中的两部分如果在执行上没有顺序规定,则这两部分程序可用线程执行。

另外,线程方式还可用于数据的批处理以及网络系统中的信息发送与接收和其他相关处理等。例如,图 3.28 给出了一个用户主机通过网络对两台远程服务器进行远程调用(RPC)以获得相应结果的执行情况。

如果用户程序只用一个线程,则第 2 个远程调用的请求只有在得到第 1 个请求的执行结果后才能发出(如图 3.28(a)所示)。

多线程时,用户程序不必等待第 1 个 RPC 请求的执行结果而直接发出第 2 个 RPC 请求(如图 3.28(b)所示),从而缩短等待时间。

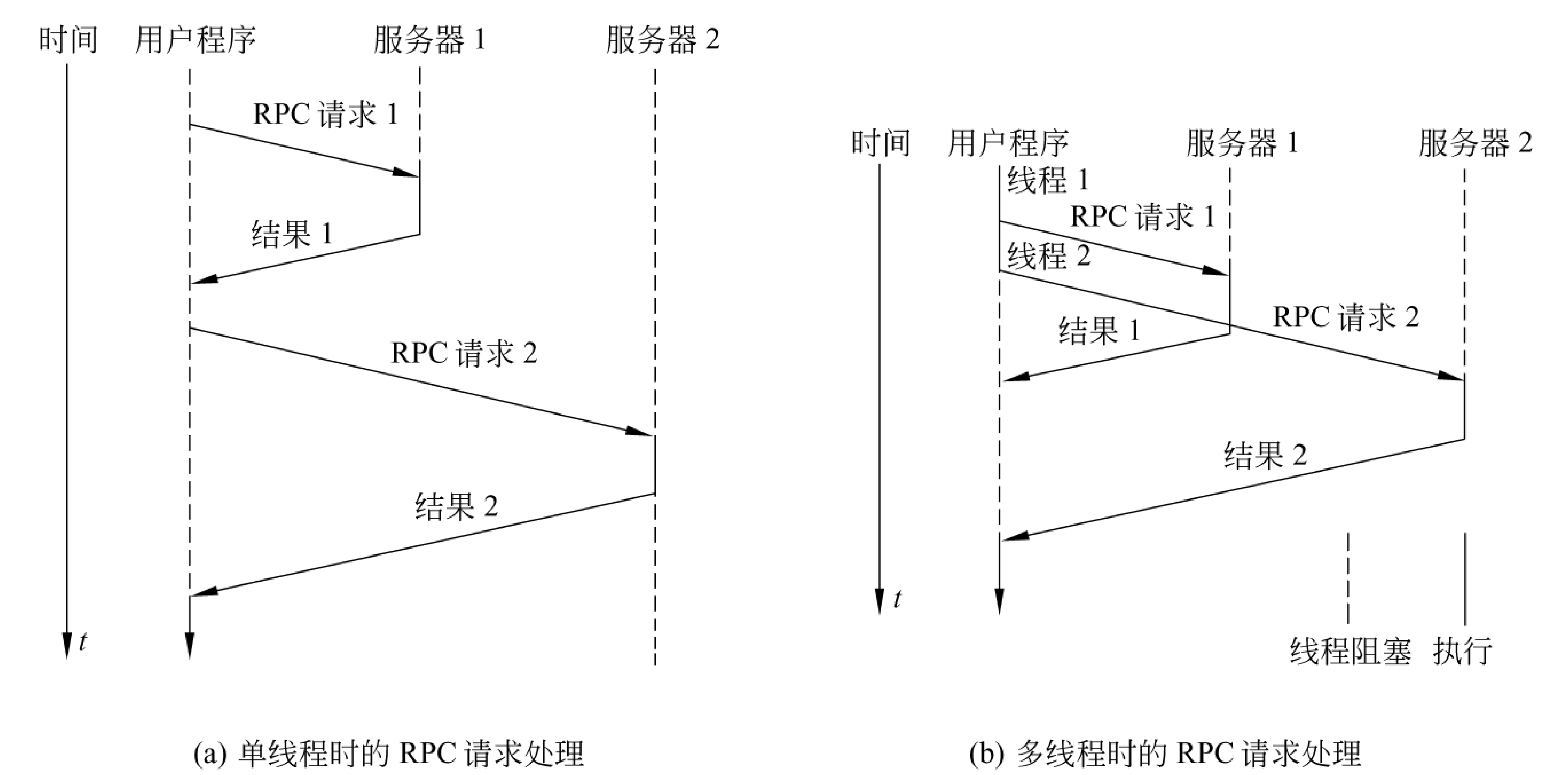



图 3.28 用户程序向两台远程服务器发出 RPC 请求的执行结果

# 3.10 线程分类与执行

## 3.10.1 线程的分类

线程的两个基本类型是用户级线程和内核级线程(系统级线程)。在同一个操作系统中,有的使用纯用户级线程,有的使用纯核心级线程,例如 Windows NT 和 OS/2 就是这样的系统;有的则混合使用用户级线程和核心级线程,例如 Solaris 操作系统。

用户级线程(user-level thread)的管理过程全部由用户程序完成,操作系统内核只对进

程进行管理。

为了对用户级线程进行管理，操作系统提供一个在用户空间执行的线程库。该线程库提供创建、调度和撤销线程功能。同时，该线程库也提供线程间的通信、线程的执行以及存储线程上下文的功能。用户级线程只使用用户堆栈和分配给所属进程的用户寄存器。当一个线程被派生时，线程库为其生成相应的线程控制块(TCB)等数据结构，并为 TCB 中的参量赋值和把该线程置于就绪状态。其处理过程与进程创建过程大致相似。

这两个过程不同的是：

(1) 用户级线程的调度算法和调度过程全部由用户自行选择和确定，与操作系统内核无关。在用户级线程系统中，操作系统内核的调度单位仍是进程。如果进程的调度区间为 $T$，则在 $T$ 区间内，用户可以根据自己的需要设置不同的线程调度算法。

(2) 用户级线程的调度算法只进行线程上下文切换而不进行处理机切换，且其线程上下文切换是在内核不参与的情况下进行的。也就是说，线程上下文切换只是在用户栈、用户寄存器等之间进行，不涉及处理机的状态。新线程通过程序调用指针的变化使得程序计数器变化而得以执行。

(3) 因为用户级线程的上下文切换与内核无关，所以可能出现如下情况：即当一个进程由于 I/O 中断或时间片用完等原因造成该进程退出处理机，而属于该进程的执行中线程仍处于执行状态。也就是说，尽管相关进程的状态是阻塞的或等待的，但所属线程状态却是执行的。

内核级线程(kernel-level thread)由操作系统内核进行管理。操作系统内核给应用程序提供相应的系统调用和应用程序接口(API)，以使用户程序可以创建、执行和撤销线程。

与用户线程不同，内核级线程既可以被调度到一个处理机上并发执行，也可以被调度到不同的处理机上并行执行。操作系统内核既负责进程的调度，也负责进程内不同线程的调度工作。因此，内核级线程不会出现进程处于阻塞或等待状态，而线程处于执行状态的情况。

另外，内核级线程技术也可用于内核程序自身，从而提高操作系统内核程序的执行效率。

内核级线程的上下文切换时间要大于用户级线程的上下文切换时间。表 3.1 给出了用户级线程、内核级线程以及进程进行上下文切换时各自的时间开销。

**表 3.1　线程、进程等的上下文切换开销**

| 操作 | 用户级线程 | 内核极线程 | 进程 |
|---|---|---|---|
| Null Fork | 34 | 948 | 11 300 |
| 信号等待 | 37 | 441 | 1840 |

表 3.1 是在 VAX 机的单处理机系统上用 UNIX 系列操作系统测试得到的结果。测试时使用了两个过程，即 Null Fork 和信号等待(signal-wait)。用户级线程、内核级线程以及进程都可用来完成上述功能。由表 3.1 可以看出，用户级线程占用的系统开销最小，内核级线程的开销则较进程开销小，但要大于用户级线程的开销。

与内核级线程相比，用户级线程的另一个优点是：实现用户级线程不需要操作系统内核的特殊支持，只要有一个能提供线程创建、调度、执行、撤销以及通信等功能的线程库就行了。

有些操作系统，例如 Linux 或 Solaris 2. x，提供用户级线程和系统级线程两种功能。在这些操作系统中，线程的创建、调度和同步等仍在用户空间完成，而这些线程也可被映射到系统空间，并转化为内核级线程执行。这与后述的 UNIX 用户进程以及系统进程有相似之处。

## 3.10.2 线程的执行特性

线程在执行时也有它的相关特性。线程的状态和同步用来反映线程的这些特性。

线程有 3 个基本状态，即执行、就绪和阻塞。但是线程没有进程中的挂起状态，也就是说，线程是一个只与内存和寄存器相关的概念，它的内容不会因交换而进入外存。

针对线程的 3 种基本状态，存在 5 种基本操作来转换线程的状态。这 5 种基本操作如下：

（1）派生（spawn）。线程在进程内派生出来，它既可由进程派生，也可由线程派生。用户一般用系统调用（或相应的库函数）派生自己的线程。例如，在 Linux 操作系统中，库函数 clone（）和 Creat-thread（）被分别用来派生不同执行模式下的线程。

一个新派生出来的线程具有相应的数据结构指针和变量，这些指针和变量作为寄存器上下文放在相应的寄存器和堆栈中。

新派生线程被放入就绪队列。

（2）阻塞（block）。如果一个线程在执行过程中需要等待某个事件发生，则被阻塞。阻塞时，寄存器上下文、程序计数器以及堆栈指针都会得到保证。

（3）激活（unblock）。如果阻塞线程的事件发生，则该线程被激活并进入就绪队列。

（4）调度（schedule）。选择一个就绪线程进入执行状态。

（5）结束（finish）。如果一个线程执行结束，它的寄存器上下文以及堆栈内容等将被释放。

线程的状态和操作关系如图 3.29 所示。

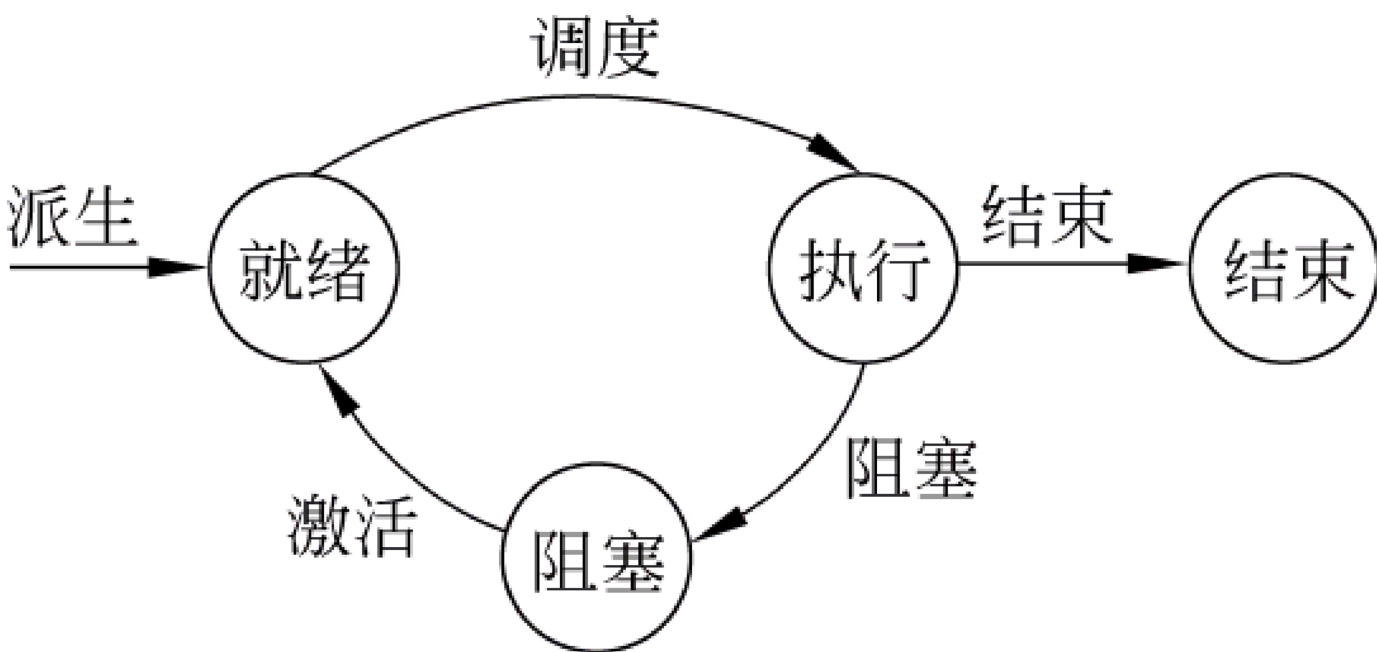


图 3.29 线程的状态与操作

需要注意的一点是，在某些情况下，某个线程被阻塞也可能导致该线程所属的进程被阻塞。

线程的另一个执行特性是同步。

由于同一进程中的所有线程共享该进程的所有资源和地址空间，任何线程对资源的操作都会对其他相关线程带来影响。因此，系统必须为线程的执行提供同步控制机制，以防止因线程的执行而破坏其他的数据结构和给其他线程带来不利的影响。

线程中所使用的同步控制机制与进程中所使用的同步控制机制相同。因此，这里不再进一步讲述有关线程的同步问题。

# 本章小结

进程是操作系统中最重要、最基本的概念之一。它是系统分配资源的基本单位，是一个具有独立功能的程序段对某个数据集的一次执行活动。引入进程的概念是由操作系统的资源有限性、处理上的并行性以及系统用户的执行起始时间的随机性所决定的。一切仅具有静态特征的概念，例如程序，不能反映系统的上述特征，因此，导入了具有动态特征的进程概念。

进程具有动态性、并发性等特点。反映进程动态特性的是进程状态的变化。进程要经历创建、等待资源、就绪准备执行、执行和执行后释放资源并消亡等几个过程和状态。进程的状态转换要由不同的原语执行完成。进程的并发特性反映在进程对资源的竞争以及由资源竞争所引起的对进程执行速度的制约。这种制约可分为直接制约和间接制约。进程间的直接制约是被制约进程和制约进程之间存在着使用对方资源的需求，只有制约进程执行后，被制约进程才能继续往前推进。进程间的间接制约是被制约进程共享某个一次只能供一个进程使用的系统资源，只有得到该资源的进程才能继续往前推进，其他进程在获得资源进程执行期间不允许交叉执行。因此，直接制约进程之间具有固定的执行顺序，而间接制约的进程之间则没有固定的执行顺序。

进程间的间接制约可利用加锁法和 P、V 原语操作实现。进程间的直接制约既可用 P、V 原语实现，也可用其他互相传递信号的方式实现。

尽管进程是一个动态概念，但是，从处理机执行的观点来看，进程仍需要静态描述。一个进程的静态描述是处理机的一个执行环境，被称为进程上下文。进程上下文由以下部分组成：PCB（进程控制块）、正文段和数据段以及各种寄存器和堆栈中的值。寄存器中主要存放将要执行指令的逻辑地址、执行模式以及执行指令时所要用到的各种调用和返回参数等。而堆栈中则存放 CPU 现场保护信息和各种资源控制管理信息等。

本章中所述的另一个重要的概念是进程通信。进程通信又可分为传送控制信号的低级通信和大量传送数据的高级通信。从通信方式来看，又可分为主从式、会话式、消息与邮箱方式以及共享虚存方式。

无论是互相通信的进程或是共享某些不同类型资源的进程，都可能因通信顺序不当或资源分配顺序不当而造成死锁。死锁是一种因各并发进程等待资源而永久不能向前推进的系统状态。死锁对操作系统是十分有害的，消除死锁的方法是预防、回避、检测与恢复 3 种。比较常用的死锁排除方法是检测与恢复方法。

线程是为了提高操作系统的执行效率而引入的，它是进程内的一段程序的基本调度单位。线程可分为用户级线程和内核级线程。用户级线程的管理全部由线程库完成，与操作系统内核无关。线程由寄存器、堆栈以及程序计数器等组成，同一进程的线程共享该进程的进程空间和其他所有资源。线程主要用于多机系统以及网络系统的操作系统中。

# 习　　题

3.1　有人说，一个进程是由伪处理机执行的一个程序，这话对吗？为什么？

3.2　试比较进程和程序的区别。

3.3 我们说程序的并发执行将导致最终结果失去封闭性。这话对所有的程序都成立吗？试举例说明。

3.4 试比较作业和进程的区别。

3.5 UNIX System Ⅴ中，系统程序所对应的正文段未被考虑成进程上下文的一部分，为什么？

3.6 什么是临界区？试举一个临界区的例子。

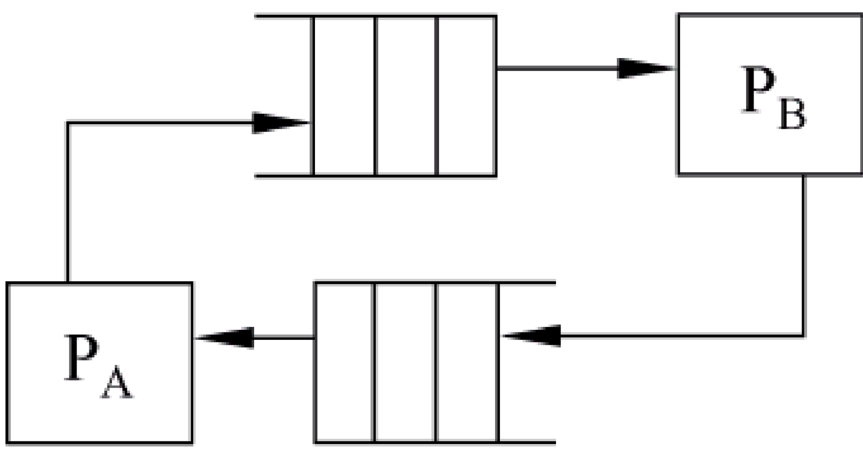
3.7 并发进程间的制约有哪两种？引起制约的原因是什么？

3.8 什么是进程间的互斥？什么是进程间的同步？

3.9 试比较 P、V 原语法和加锁法实现进程间互斥的区别。

3.10 设在 3.6 节中所描述的生产者-消费者问题中，其缓冲部分由 $m$ 个长度相等的有界缓冲区组成，且每次传输数据长度等于有界缓冲区长度，生产者和消费者可对缓冲区同时操作。重新描述发送过程 deposit(data)和接收过程 remove(data)。

3.11 两个进程 $P_A$ 和 $P_B$ 通过两个 FIFO 缓冲区队列连接(如下图所示)，每个缓冲区长度等于传送消息长度。进程 $P_A$ 和 $P_B$ 之间的通信满足如下条件：




(1) 至少有一个空缓冲区存在时，相应的发送进程才能发送一个消息。

(2) 当缓冲队列中至少存在一个非空缓冲区时，相应的接收进程才能接收一个消息。

试描述发送过程 send(i,m)和接收过程 receive(i,m)。这里 i 代表缓冲队列。

3.12 在和控制台通信的例子中，设操作员不仅回答用户进程所提出的问题，而且还能独立地向各用户进程发出指示。对于这些指示，操作员不要求用户进程回答，但它们享有比其他消息优先传送的优先度。即如果 inbuf 中有指示存在，系统不能进行下一次通信会话。试按上述要求重新描述 CCP、KCP 和 DCP。

3.13 编写一个程序使用系统调用 fork 生成 3 个子进程，并使用系统调用 pipe 创建一个管道，使得这 3 个子进程和父进程公用同一管道进行信息通信。

3.14 设有 5 个哲学家，共享一张放有 5 把椅子的桌子，每人分得一把椅子。但是，桌子上总共只有 5 支筷子，在每人两边分开各放一支。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。条件如下：

(1) 只有拿到两支筷子时，哲学家才能吃饭。

(2) 如果筷子已在他人手上，则该哲学家必须等待到他人吃完之后才能拿到筷子。

(3) 任一哲学家在自己未拿到两支筷子吃饭之前，决不放下自己手中的筷子。

试解答以下问题：

(1) 描述一个保证不会出现两个邻座同时要求吃饭的通信算法。

(2) 描述一个既没有两邻座同时吃饭，又没有人饿死(永远拿不到筷子)的算法。

(3) 在什么情况下，5 个哲学家全部吃不上饭？

3.15 什么是线程？试述线程与进程的区别。

3.16 使用库函数 clone()与 creat-thread()在 Linux 环境下创建两种不同执行模式的线程程序。

# 第4章　处理机调度

CPU是计算机系统中十分重要的资源。但在早期的计算机系统中,对它的管理是十分简单的。因为那时它和其他系统资源一样,为一个作业所独占,不存在处理机分配和调度问题。随着多道程序设计技术和各种不同类型的操作系统的出现,各种不同的CPU管理方法得到启用。不同的CPU管理方法将为用户提供不同性能的操作系统。例如,在多道批处理系统中,为了提高处理机的效率和增加作业吞吐率,当调度一批作业组织多道运行时,要尽可能使作业搭配合理,充分利用系统中的各种资源。在分时系统中,由于用户使用交互式会话的工作方式,系统必须要有较快的响应时间,使得每个用户都感到如同只有自己一人在使用这台计算机。因此,系统在调度作业执行时,首先考虑的是每个用户作业得到处理机的均等性。在实时系统中,首先要考虑的是处理机的响应时间。由此可见,操作系统的要求不同,处理机管理的策略是不同的。

衡量调度策略的指标很多。最常用的几个指标是周转时间、吞吐率、响应时间以及设备利用率等。

周转时间是指将一个作业提交给计算机系统后到该作业的结果返回给用户所需要的时间。

吞吐率是指在给定的时间内,一个计算机系统所完成的总工作量。

响应时间则是指从用户向计算机发出一个命令到计算机把相应的执行结果返回给用户所需要的时间。

设备利用率主要指输入输出设备的使用情况,在有些要求I/O处理能力强(如管理信息系统)的系统中,高的设备利用率也是衡量调度策略好坏的一个重要指标。

本章将以CPU管理为核心,讨论管理和控制用户进程执行的方法。主要包括以下内容:

(1) 作业与进程的关系;

(2) 作业调度策略与算法;

(3) 进程调度策略与算法;

(4) 几种调度策略的评价。

另外,本章还介绍实时调度系统。

## 4.1　分级调度

### 4.1.1　作业的状态及其转换

第2章介绍了作业的概念。作业是用户要求计算机所做的关于一次业务处理的全部工作,它包括作业的提交、执行和输出等过程。一个作业从用户提交开始到占有处理机被执行,要由系统经过多级调度才能实现(在有些系统,例如分时系统中,也可以由单级调度实

现)。下面以批处理系统为例看一个作业处理的大致过程。

如图 4.1 所示,一个作业从提交给计算机系统到执行结束退出系统,一般都要经历提交、收容、执行和完成 4 个状态。

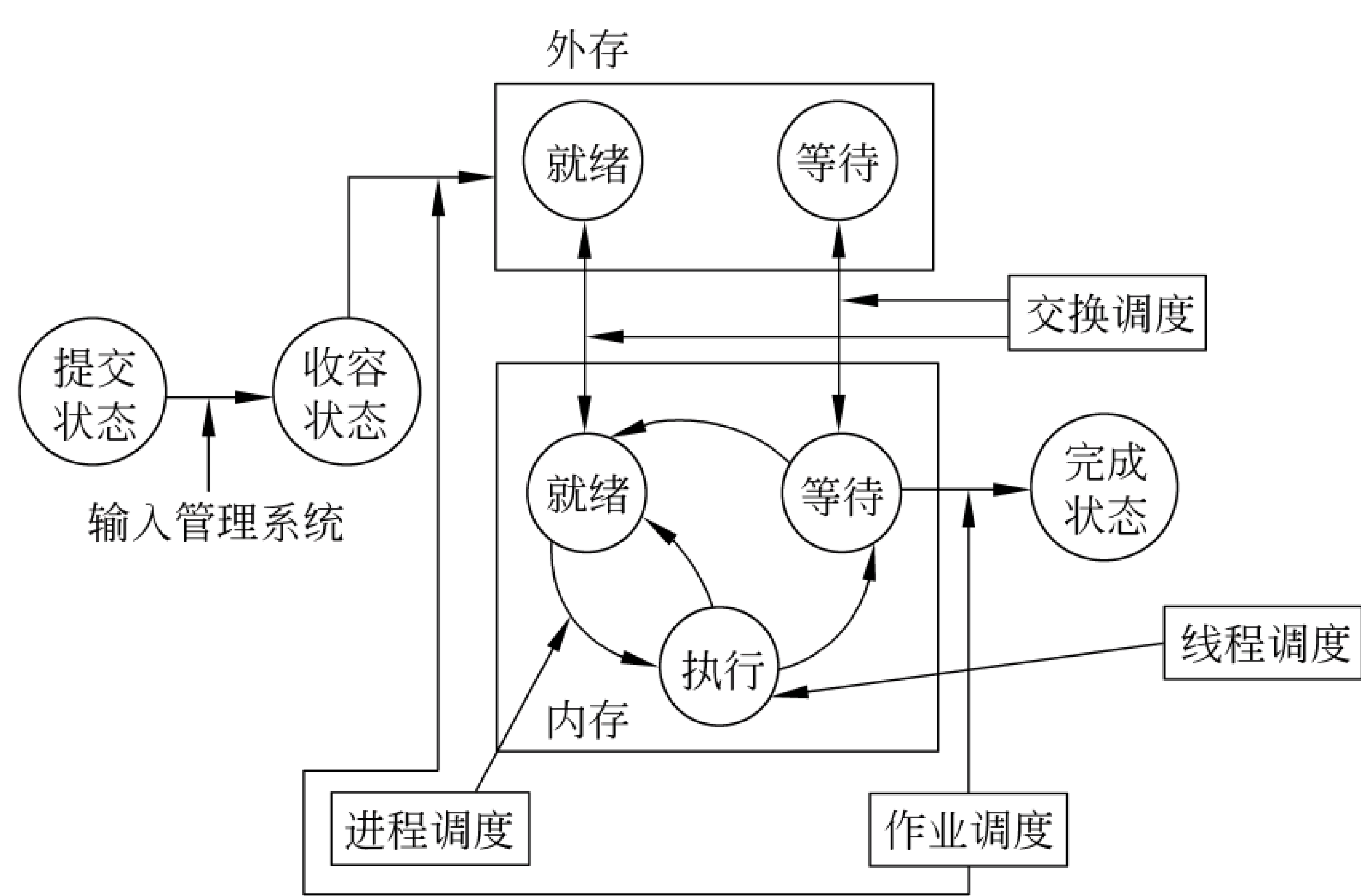



图 4.1 作业的状态及其转换

一个作业在其处于从输入设备进入外部存储设备的过程称为提交状态。处于提交状态的作业,因其信息尚未全部进入系统,所以不能被调度程序选取。

收容状态也称为后备状态。输入管理系统不断地将作业输入到外存中的对应部分(或称输入井,即专门用来存放待处理作业信息的一组外存分区)。若一个作业的全部信息已全部被输入进输入井,那么,在它还未被调度去执行之前,该作业处于收容状态。

作业调度程序从后备作业中选取若干个作业到内存投入运行。它为被选中作业建立进程并分配必要的资源,这时,这些被选中的作业处于执行状态。从宏观上看,这些作业正处在执行过程中;但从微观上看,在某一时刻,由于处理机总数少于并发执行的进程数,因此,不是所有被选中作业都占有处理机,其中的大部分处于等待资源或就绪状态中。那么,究竟哪个作业的哪个进程能获得处理机而真正在执行,要依靠进程调度来决定。

当作业运行完毕,但它所占用的资源尚未全部被系统回收时,该作业处于完成状态。在这种状态下,系统需做诸如打印结果、回收资源等善后处理工作。

## 4.1.2 调度的层次

处理机调度问题实际上也是处理机的分配问题。哪些作业的哪些进程可以参加竞争处理机呢?显然,只有那些参与竞争处理机所必需的资源都已得到满足的进程才能享有竞争处理机的资格。这时,它们处于内存就绪状态。这些必需的资源包括内存、外设及有关数据结构等。从而,在进程有资格竞争处理机之前,作业调度程序必须先调用存储管理和外设管理程序,并按一定的选择顺序和策略从输入井中选择出几个处于后备状态的作业,为它们分配内存等资源和创建进程,使它们获得竞争处理机的资格。

另外,由于处于执行状态下的作业一般包含多个进程,而在单机系统中,每一时刻只能有一个进程占有处理机。那么,其他进程就只能处于准备抢占处理机的就绪状态或等待得到某种新资源的等待状态。为了提高资源的利用率,在有些操作系统中把一部分在内存中

处于就绪状态或等待状态而在短时期内又得不到执行的进程、作业换出内存,以让其他作业的进程竞争处理机。这样,在外存中,除了处于后备状态的作业外,还存在处于就绪状态而等待得到内存的作业。这就需要有一定的方法和策略为这部分作业分配空间。

一般来说,处理机调度可以分为 4 级:

(1) 作业调度。又称宏观调度或高级调度。其主要任务是按一定的原则对外存输入井上的大量后备作业进行选择,给选出的作业分配内存和输入输出设备等必要的资源,并建立相应的根进程,以使该作业的进程获得竞争处理机的权利。另外,当该作业执行完毕时,还负责回收系统资源。

(2) 交换调度。又称中级调度。其主要任务是按照给定的原则和策略,将处于外存交换区中的就绪状态或等待状态的进程调入内存,或把处于内存就绪状态或内存等待状态的进程交换到外存交换区。交换调度主要涉及内存管理与扩充,因此,在有些书中也把它归入内存管理部分。

(3) 进程调度。又称微观调度或低级调度。其主要任务是按照某种策略和方法选取一个处于就绪状态的进程占用处理机。在确定了占用处理机的进程之后,系统必须进行进程上下文切换以建立与占用处理机进程相适应的执行环境。

(4) 线程调度。上述 4 级调度的关系如图 4.1 所示。

在多道批处理系统中,存在着作业调度和进程调度。但是,在分时系统和实时系统中,一般不存在作业调度,而只有进程调度、交换调度和线程调度。这是因为在分时系统和实时系统中,为了缩短响应时间或为了满足用户需求的截止时间,作业不是建立在外存,而是直接建立在内存中。在这些系统中,一旦用户和系统的交互开始,用户马上要进行控制。因而,这些系统中没有作业提交状态和后备状态。它们的输入信息经过终端缓冲区为系统所接收,或者立即处理,或者经交换调度暂存于外存中。

### 4.1.3 作业与进程的关系

作业可被看作是用户向计算机提交任务的任务实体,例如一次计算和一个控制过程等。反过来,进程则是计算机为了完成用户任务而设置的执行实体,是系统分配资源的基本单位。显然,计算机要完成一个任务实体,必须要有一个以上的执行实体。也就是说,一个作业总是由一个以上的进程组成。

作业怎样分解为进程呢? 首先,系统必须为一个作业创建一个根进程;然后,在执行作业控制语句时,根据任务要求,系统或根进程为其创建相应的子进程;最后,为各子进程分配资源和调度各子进程执行以完成作业要求的任务。

## 4.2 作业调度

如上所述,作业调度主要是完成作业从后备状态到执行状态的转变,以及从执行状态到完成状态的转变。本节主要介绍作业调度的功能及调度性能的评价方法。

### 4.2.1 作业调度功能

作业调度有以下主要功能。

（1）记录系统中各作业的状况，包括执行阶段的有关情况。通常，系统为每个作业建立一个作业控制块(JCB)记录这些有关信息。与系统要通过进程控制块(PCB)感知进程存在一样，系统通过JCB而感知作业的存在。系统在作业进入后备状态时为该作业建立它的JCB，从而使得该作业可被作业调度程序感知。当该作业执行完毕进入完成状态之后，系统又撤销其JCB而释放有关资源并撤销该作业。每个作业在各阶段所要求和分配的资源以及该作业的状态都记录在它的JCB中，根据JCB中的有关信息，作业调度程序对作业进行调度和管理。

| 作业名 |
| --- |
| 作业类型 |
| 资源要求 |
| 资源使用情况 |
| 优先级（数） |
| 当前状态 |
| 其他 |

图4.2　作业控制块(JCB)

对于不同的批处理系统，其JCB的内容也有所不同。图4.2给出了JCB的主要内容。它包括作业名、作业类型、资源要求、当前状态、资源使用情况以及该作业的优先级等。

其中，作业名由用户提供并由系统将其转换为系统可识别的作业标识符。作业类型指该作业属于计算型(要求CPU时间多)还是管理型(要求输入输出量大)，或图形设计型(要求高速图形显示)等。而资源要求则包括该作业估计执行时间、要求的最迟完成时间、要求的内存量和外存量、要求的外设类型及台数以及要求的软件支持工具库函数等。资源要求均由用户提供。资源使用情况包括作业进入系统时间、开始执行时间、已执行时间、内存地址和外设台数等。作业进入系统时间指作业的全部信息进入输入井，作业的状态成为后备状态的时间。开始执行时间指该作业被调度程序选中，其状态由后备状态变为执行状态的时间。内存地址指分配给该作业的内存区起始地址。外设台数指分配给该作业的外设实际台数。优先级则用来决定该作业的调度次序。优先级既可以由用户给定，也可以由系统动态计算产生。当前状态是指该作业当前所处的状态。显然，只有当作业处于后备状态时，该作业才可以被调度。

（2）从后备队列中挑选出一部分作业投入执行。一般来说，系统中处于后备状态的作业较多，大的系统可以达到几十个甚至几百个。这取决于输入井的空间大小。但是，处于执行状态的作业一般只有有限的几个。作业调度程序根据选定的调度算法，从后备作业队列中挑选出若干作业去投入执行。

（3）为被选中作业做好执行前的准备工作。作业调度程序为选中的作业建立相应的进程，并为这些进程分配它们所需要的系统资源，如分配给它们内存、外存和外设等。

（4）在作业执行结束时做善后处理工作。主要是输出作业管理信息，例如执行时间等。再就是回收该作业所占用的资源，撤销与该作业有关的全部进程和该作业的作业控制块等。

作业从后备状态到执行状态，又从执行状态到完成状态的转换过程如图4.3所示。

## 4.2.2　作业调度目标与性能衡量

现在已经知道，作业调度的功能是：记录系统中各作业的状况；从后备作业队列中挑选一批作业进入执行状态；为被选中作业分配资源建立进程，并在作业执行结束后释放所占用的资源等。其中最主要的是从后备作业队列中选取一批作业进入执行状态。怎样挑选呢？显然，根据不同的目标，会有不同的调度算法。这些调度算法将在4.4节中介绍，这里先介绍调度目标。

一般来说，调度目标主要是以下4点：

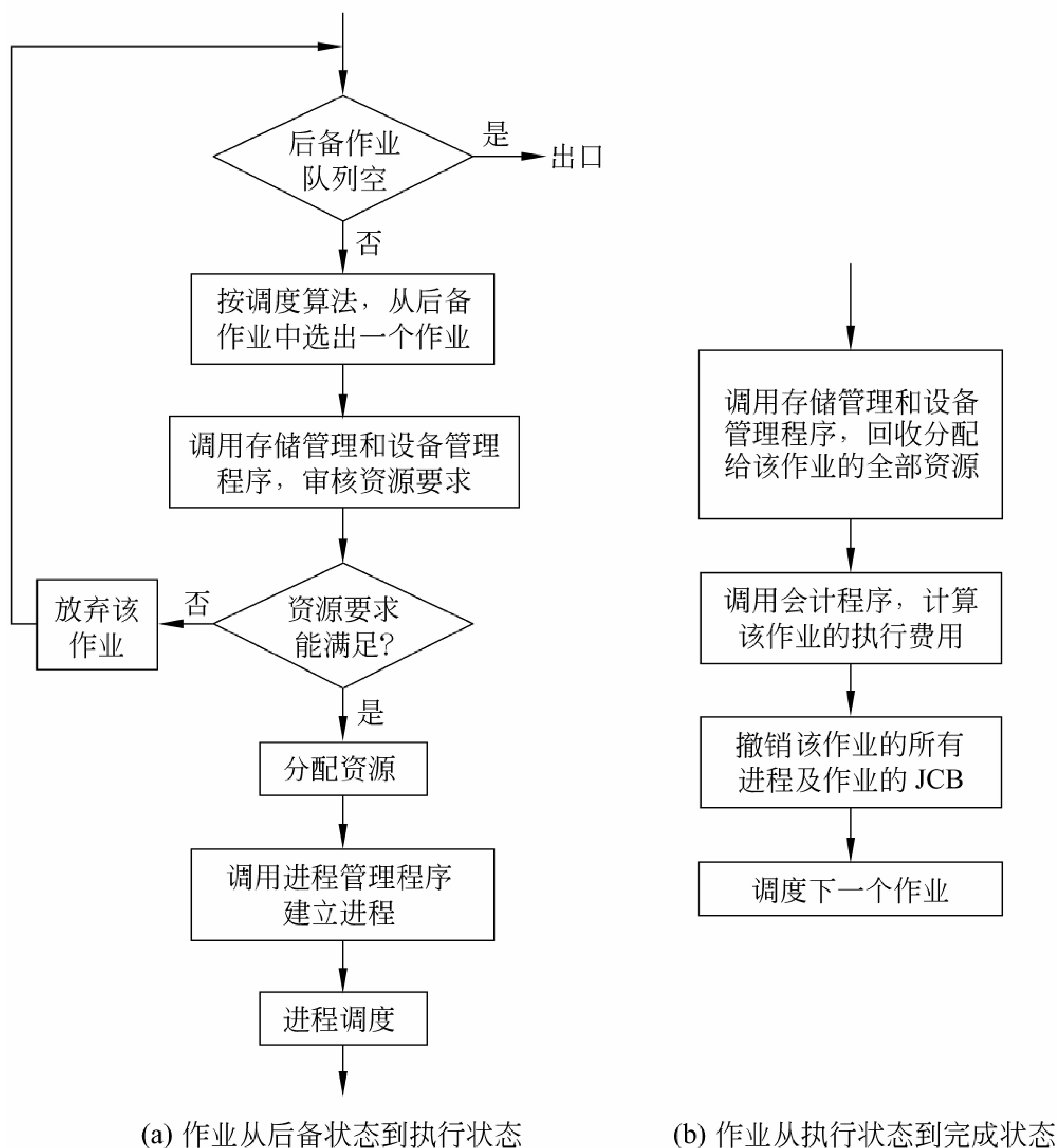



图 4.3　作业调度中状态的转换过程

（1）对所有作业应该是公平合理的。

（2）应使设备有高的利用率。

（3）每天执行尽可能多的作业。

（4）有快的响应时间。

由于这些目标的相互冲突，任一调度算法要想同时满足上述目标是不可能的。例如，要想执行尽可能多的作业，调度算法就应选择那些估计执行时间短的作业，但这样做对那些估计执行时间长的作业又是不公平的，使它们的响应时间将会变得非常长。

必须指出，如果考虑的因素过多，调度算法就会变得非常复杂。其结果是系统开销增加，资源利用率下降。因此，大多数操作系统都根据用户需要，采用兼顾某些目标的简单调度算法。

那么，怎样来衡量一个作业调度算法是否满足系统设计的要求呢？对于批处理系统，由于主要用于计算，对于作业的周转时间要求较高。因此，作业的平均周转时间或平均带权周转时间被作为衡量调度算法优劣的标准。但是，对于分时系统和实时系统来说，另外增加了平均响应时间作为衡量调度策略优劣的标准。

**1. 周转时间**

作业 $i$ 的周转时间 $T_i$ 为

$$T_i = Te_i - Ts_i$$

其中 $Te_i$ 为作业 $i$ 的完成时间，$Ts_i$ 为作业的提交时间。

对于被测定作业流所含有的 $n(n \geqslant 1)$ 个作业来说，其平均周转时间为

$$T = \frac{1}{n}\sum_{i=1}^{n} T_i$$

一个作业的周转时间说明了该作业在系统内停留的时间，包含两部分：一为等待时间，二为执行时间，即

$$T_i = Tw_i + Tr_i$$

这里，$Tw_i$ 主要指作业 $i$ 由后备状态到执行状态的等待时间，它不包括作业进入执行状态后的等待时间。

**2. 带权周转时间**

作业的周转时间包含了两个部分，即等待时间和执行时间。为了进一步反映调度性能，引入了带权周转时间的概念。带权周转时间是作业周转时间与作业执行时间的比：

$$W_i = T_i / Tr_i$$

对于被测定作业流所含有的几个作业来说，其平均带权周转时间为

$$W = \frac{1}{n}\sum_{i=1}^{n} W_i$$

对于分时系统，除了要保证系统吞吐量大、资源利用率高之外，还应保证有用户能够容忍的响应时间。因此，在分时系统中，仅仅用周转时间或带权周转时间来衡量调度性能是不够的。

# 4.3 进程调度

无论是在批处理系统、分时系统还是实时系统中，用户进程数一般都多于处理机数，这将导致用户进程互相争夺处理机。另外，系统进程也同样需要使用处理机。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程，以使之执行。本节介绍进程调度的功能和进程调度发生的时机等。

## 4.3.1 进程调度的功能

进程调度的具体功能可总结如下：

(1) 记录系统中所有进程的执行情况。

作为进程调度的准备，进程管理模块必须将系统中各进程的执行情况和状态特征记录在各进程的 PCB 中，并且，进程管理模式根据各进程的状态特征和资源需求，将各进程的 PCB 排成相应的队列并进行动态队列转接。进程调度模块通过 PCB 变化来掌握系统中所有进程的执行情况和状态特征，并在适当的时机从就绪队列中选择出一个进程占据处理机。

(2) 选择占有处理机的进程。

进程调度的主要功能是按照一定的策略选择一个处于就绪状态的进程，使其获得处理机执行。根据不同的系统设计目的，有各种各样的选择策略，例如系统开销较小的静态优先数调度法，适用于分时系统的轮转法(round robin)和多级反馈轮转法(round robin with

multiple feedback)等。这些选择策略决定了调度算法的性能。有关这些算法，将在 4.5 节中描述。

（3）进行进程上下文切换。

当正在执行的进程由于某种原因要让出处理机时，系统要做进程上下文切换，以使被调度选中的进程得以执行。被选中进程必须从上一次被中断处开始执行。这就要恢复该进程的上下文和进行上下文切换，系统在做上下文切换时，首先要检查是否可以做上下文切换（在有些情况下，上下文切换是不允许的，例如系统正在执行某个不允许中断的原语时）。然后，系统要保留有关被切换进程的足够信息，以便以后切换回该进程时顺利恢复该进程的执行。在系统保留了 CPU 现场之后，调度程序选择一个新的处于就绪状态的进程，并装配成该进程的上下文，使 CPU 的控制权转换到被选中进程中。

### 4.3.2 进程调度的时机

进程调度发生在什么时机呢？这与引起进程调度的原因以及进程调度的方式有关。

引起进程调度的原因有以下几类：

（1）正在执行的进程执行完毕。这时，如果不选择新的就绪进程执行，将浪费处理机资源。

（2）执行中进程自己调用阻塞原语将自己阻塞起来进入睡眠等待状态。

（3）执行中进程调用了 P 原语操作，从而因资源不足而被阻塞；或调用了 V 原语操作激活了等待资源的进程队列。

（4）执行中进程提出 I/O 请求后被阻塞。

（5）在分时系统中时间片已经用完。

（6）在执行完系统调用，在系统程序返回用户进程时，可认为系统进程执行完毕，从而可调度选择一个新的用户进程执行。

以上都是在 CPU 执行不可剥夺方式下所引起进程调度的原因。在 CPU 执行方式是可剥夺时，还有以下的原因：

（7）就绪队列中的某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

所谓可剥夺方式，即就绪队列中一旦有优先级高于当前执行进程优先级的进程存在时，便立即发生进程调度，转让处理机。而非剥夺方式或不可剥夺方式意味着：即使在就绪队列存在有优先级高于当前执行进程时，当前进程仍将继续占有处理机，直到该进程自己因调用原语操作或等待 I/O 而进入阻塞、睡眠状态，或时间片用完时才重新发生调度让出处理机。

操作系统将在以上几种原因之一发生的情况下进行进程调度。例如，UNIX System Ⅴ 就是在以下 5 种情况之一发生时进行进程调度的：

（1）当前进程自己调用 sleep 和 wait 等进入睡眠状态时。

（2）当前进程从系统调用执行结束后返回用户态时，它的优先级已低于其他就绪状态进程，或调度标志被置位。

（3）当前进程在完成中断和陷阱处理后返回用户态时，它的优先级已低于其他就绪状态进程，或调度标志被置位。

(4) 时间片用完,而且当前进程的优先级低于其他就绪进程。

(5) 当前进程调用 exit 自我终止时。

### 4.3.3 进程调度性能评价

进程调度虽然是系统内部的低级调度,但进程调度的优劣直接影响作业调度的性能。那么,怎样评价进程调度的优劣呢?反映作业调度优劣的周转时间和平均周转时间只在某种程度上反映了进程调度的性能,例如,其执行时间部分中实际上包含有进程等待(包括就绪状态时的等待)时间,而进程等待时间的多少是要依靠进程调度策略和等待事件何时发生来决定的。因此,进程调度性能的衡量是操作系统设计的一个重要指标。

进程调度性能的衡量方法可分为定性和定量两种。在定性衡量方面,首先是调度的可靠性。包括一次进程调度是否可能引起数据结构的破坏等。这要求对调度时机的选择和保存 CPU 现场十分谨慎。另外,简洁性也是衡量进程调度的一个重要指标,由于调度程序的执行涉及多个进程和必须进行上下文切换,如果调度程序过于烦琐和复杂,将会耗去较大的系统开销。这在用户进程调用系统调用较多的情况下,将会造成响应时间大幅度增加。

进程调度的定量评价包括 CPU 的利用率评价、进程在就绪队列中的等待时间与执行时间之比等。实际上,由于进程进入就绪队列的随机模型很难确定,而且进程上下文切换等也将影响进程的执行效率,从而对进程调度进行解析是很困难的。一般情况下,大多利用模拟或测试系统响应时间的方法来评价进程调度的性能。

## 4.4 调度算法

本节讨论各种常用的进程调度算法和作业调度算法。

**1. 先来先服务(FCFS)调度算法**

将用户作业和就绪进程按提交顺序或变为就绪状态的先后排成队列,并按照先来先服务(First Come First Serve,FCFS)的方式进行调度处理,是一种最普遍和最简单的方法。在没有特殊理由要优先调度某类作业或进程时,从处理的角度来看,FCFS 方式是一种最合适的方法,因为无论是追加还是取出一个队列元素在操作上都是最简单的。

直观看,该算法在一般意义下是公平的。即每个作业或进程都按照它们在队列中等待的时间长短来决定它们是否优先享受服务。不过对于那些执行时间较短的作业或进程来说,如果它们在某些执行时间很长的作业或进程之后到达,则它们将等待很长的时间。

在实际的操作系统中,尽管很少单独使用 FCFS 算法,但和其他一些算法配合起来,FCFS 算法还是使用得相当多的。例如,基于优先级的调度算法就是对具有同样优先级的作业或进程采用的 FCFS 方式。

**2. 轮转法(round robin)**

轮转法的基本思路是让每个进程在就绪队列中的等待时间与享受服务的时间成比例。轮转法的基本概念是将 CPU 的处理时间分成固定大小的时间片。如果一个进程在被调度选中之后用完了系统规定的时间片,但未完成要求的任务,则它自行释放自己所占有的 CPU 而排到就绪队列的末尾,等待下一次调度。同时,进程调度程序又去调度当前就绪队列中的第一个进程或作业。轮转法的原理见图 4.4。

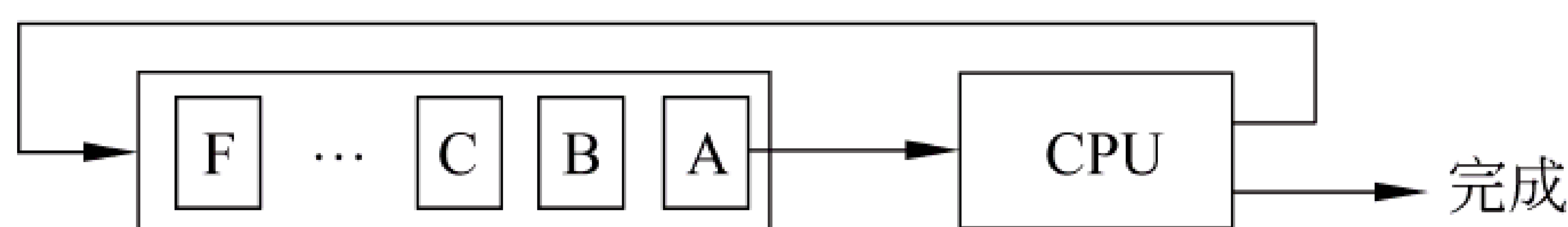



图 4.4　轮转法调度

显然，轮转法只能用来调度分配那些可以抢占的资源。将它们随时剥夺再分配给别的进程。CPU 是可抢占资源的一种。但如打印机等资源是不可抢占的。由于作业调度是对除了 CPU 之外的所有系统硬件资源的分配，其中包含了不可抢占资源，所以作业调度不使用轮转法。

在轮转法中，时间片长度的选取非常重要。首先，时间片长度的选择会直接影响系统开销和响应时间。如果时间片长度过短，则调度程序剥夺处理机的次数增多，这将使进程上下文切换次数也大大增加，从而加重系统开销。反过来，如果时间片长度选择过长，比方说一个时间片能保证就绪队列中所需执行时间最长的进程能执行完毕，则轮转法变成了先来先服务法。

时间片长度 $q$ 的选择是根据系统对响应时间的要求 $R$ 和就绪队列中所允许的最大进程数 $N_{max}$ 确定的。它可表示为

$$q = R/N_{max}$$

在 $q$ 为常数的情况下，如果就绪队列中的进程数发生远小于 $N_{max}$ 的变化，则响应时间 $R$ 看上去会大大减小。但是，就系统开销来说，由于 $q$ 值固定，从而进程上下文切换的时机不变，系统开销也不变。通常，系统开销也是处理机执行时间的一部分。CPU 的整个执行时间等于各进程执行时间加上系统开销。在进程执行时间大幅度减少的情况下，如果系统开销也随之减少的话，系统的响应时间有可能更好一点。例如，在一个用户进程的情况下，如果 $q$ 值增大到足够该进程执行完毕的话，则进程调度所引起的系统开销就没有了。一种可行的办法是，每当一轮调度开始时，系统便根据就绪队列中已有进程数目计算一次 $q$ 值，作为新一轮调度的时间片长度。这种方法得到的时间片是随就绪队列中的进程数变化的。

**3. 多级反馈轮转法(round robin with multiple feedback)**

在轮转法中，加入到就绪队列的进程有 3 种情况，一种是分给它的时间片用完，但进程还未完成，回到就绪队列的末尾等待下次调度再继续执行。第二种情况是分给该进程的时间片并未用完，只是因为请求 I/O 或由于进程的互斥与同步关系而被阻塞。当阻塞解除之后再回到就绪队列。第三种情况就是新创建进程进入就绪队列。如果对这 3 种进程区别对待，给予不同的优先级和时间片，从直观上看，可望进一步改善系统服务质量和效率。例如，可按照进程到达就绪队列的类型和进程被阻塞时的阻塞原因分成不同的就绪队列，每个队列按 FCFS 原则排列，各队列之间的进程享有不同的优先级，但同一队列内优先级相同。这样，当一个进程在执行完它的时间片，或从睡眠中被唤醒以及被创建之后，将进入不同的就绪队列。多级反馈轮转法与优先级法在原理上的区别是，一个进程在执行结束之前，可能需要反复多次通过反馈循环执行，而不是优先级法中的一次执行。

**4. 优先级法**

优先级法可被用作作业或进程的调度策略。系统或用户首先按某种原则为作业或进程指定一个优先级来表示该作业或进程所享有的调度优先权。该算法的核心是确定进程或作业的优先级。

确定优先级的方法可分为两类，即静态法和动态法。静态法根据作业或进程的静态特性，在作业或进程开始执行之前就确定它们的优先级，一旦开始执行之后就不能改变。动态法则不然，它把作业或进程的静态特性和动态特性结合起来确定作业或进程的优先级，随着作业或进程的执行过程，其优先级不断变化。

1）静态优先级

作业调度中的静态优先级大多按以下原则确定：

（1）由用户自己根据作业的紧急程度输入一个适当的优先级。为防止各用户都将自己的作业冠以高优先级，系统应对高优先级用户收取较高的费用。

（2）由系统或操作员根据作业类型指定优先级。作业类型一般由用户约定或由操作员指定。例如，可将作业分为

- I/O 繁忙的作业；
- CPU 繁忙的作业；
- I/O 与 CPU 均衡的作业；
- 一般作业，等等。

系统或操作员可以给每类作业指定不同的优先级。

（3）系统根据作业要求资源情况确定优先级。例如，根据估计所需处理机时间、内存大小、I/O 设备类型及数量等确定作业的优先级。

进程的静态优先级确定可以采用以下原则：

（1）按进程的类型给予不同的优先级。例如，在有些系统中，进程被划分为系统进程和用户进程。系统进程享有比用户进程高的优先级。对于用户进程来说，则可以分为

- I/O 繁忙的进程；
- CPU 繁忙的进程；
- I/O 与 CPU 均衡的进程；
- 其他进程。

对系统进程，也可以根据其所要完成的功能划分为不同的类型，例如调度进程、I/O 进程、中断处理进程和存储管理进程等。这些进程还可进一步划分为不同类型和赋予不同的优先级。例如，在操作系统中，对于键盘中断的处理优先级和对于电源掉电中断的处理优先级是不相同的。

（2）将作业的静态优先级作为它所属进程的优先级。

2）动态优先级

基于静态优先级的调度算法实现简单，系统开销小，但由于静态优先级一旦确定之后，直到执行结束为止始终保持不变，从而系统效率较低，调度性能不高。现在的操作系统中，如果使用优先级调度的话，则大多采用动态优先级的调度策略。

进程的动态优先级一般根据以下原则确定：

（1）根据进程占有 CPU 时间的长短来决定。一个进程占有处理机的时间越长，则在被阻塞之后再次获得调度的优先级就越低，反之，其获得调度的可能性就会越大。

（2）根据就绪进程等待 CPU 的时间长短来决定。一个就绪进程在就绪队列中等待的时间越长，则它获得调度的优先级就越高。

由于动态优先级随时间的推移而变化，系统要经常计算各进程的优先级，因此，系统要

为此付出一定的开销。

下面介绍线性优先级调度策略(Selfish Round Robin,SRR)。

使用轮转法调度进程时,新创建的进程也放入就绪队列末尾享受平等的处理机时间片。这对于执行时间长的进程来说是有点不公平的,因为它们需要多个时间片才能完成。因此,线性优先级调度策略采用如下方式,即新创建的进程按 FCFS 方式排成就绪队列,而其他已得到过时间片服务的进程也按 FCFS 方式排成另一个就绪队列或称享受服务队列(见图 4.5)。

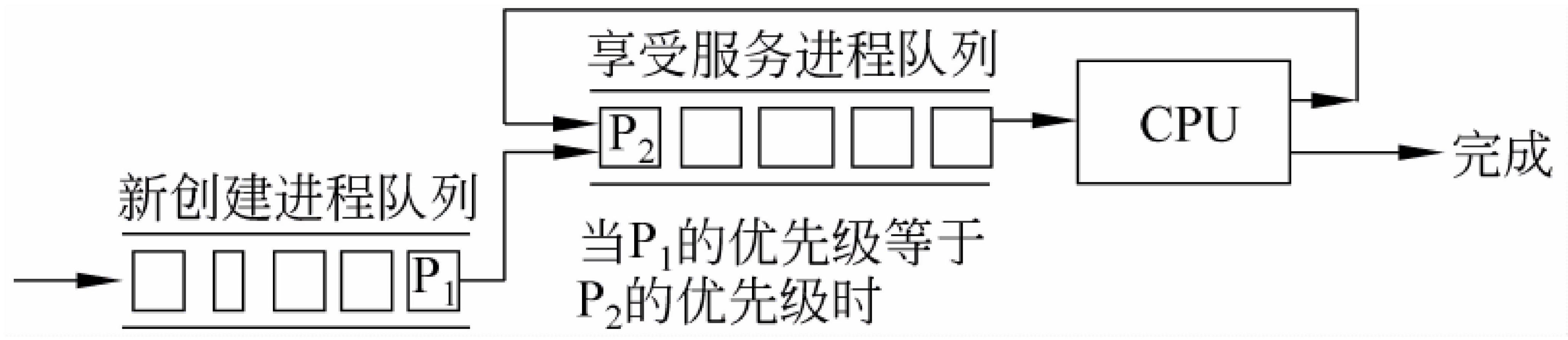



图 4.5　线性优先级调度

对于这两个不同队列中的进程,设新创建进程就绪队列中进程的优先级 $P$ 以

$$P = a * t \quad (a > 0)$$

的速率增加。另外,享受服务队列中进程的优先级 $P$ 以

$$P = b * t \quad (a > b > 0)$$

的速率增长。设某一进程在时刻 $t_1$ 时被创建,在时刻 $t$ 时,该进程的优先级为

$$P(t) = a * (t - t_1) \quad (t_1 < t < t'_1)$$

又设该进程在 $t'_1$ 时刻转入享受服务队列,则在时刻 $t$,该进程的优先级变为

$$P(t) = a * (t'_1 - t_1) + b * (t - t'_1) \quad (t'_1 < t < t'_2)$$

那么,一个新创建进程等待多长时间之后进入享受服务队列较为合适呢?当新创建进程就绪队列中的头一个进程的优先级 $P(t) = a * (t - t_1)$ 与享受服务队列中最后一个就绪进程的优先级 $P(t) = b * t$ 相等时,新创建进程队列中的头一个进程可以转入享受服务进程队列。其优先级变化曲线如图 4.6 所示。

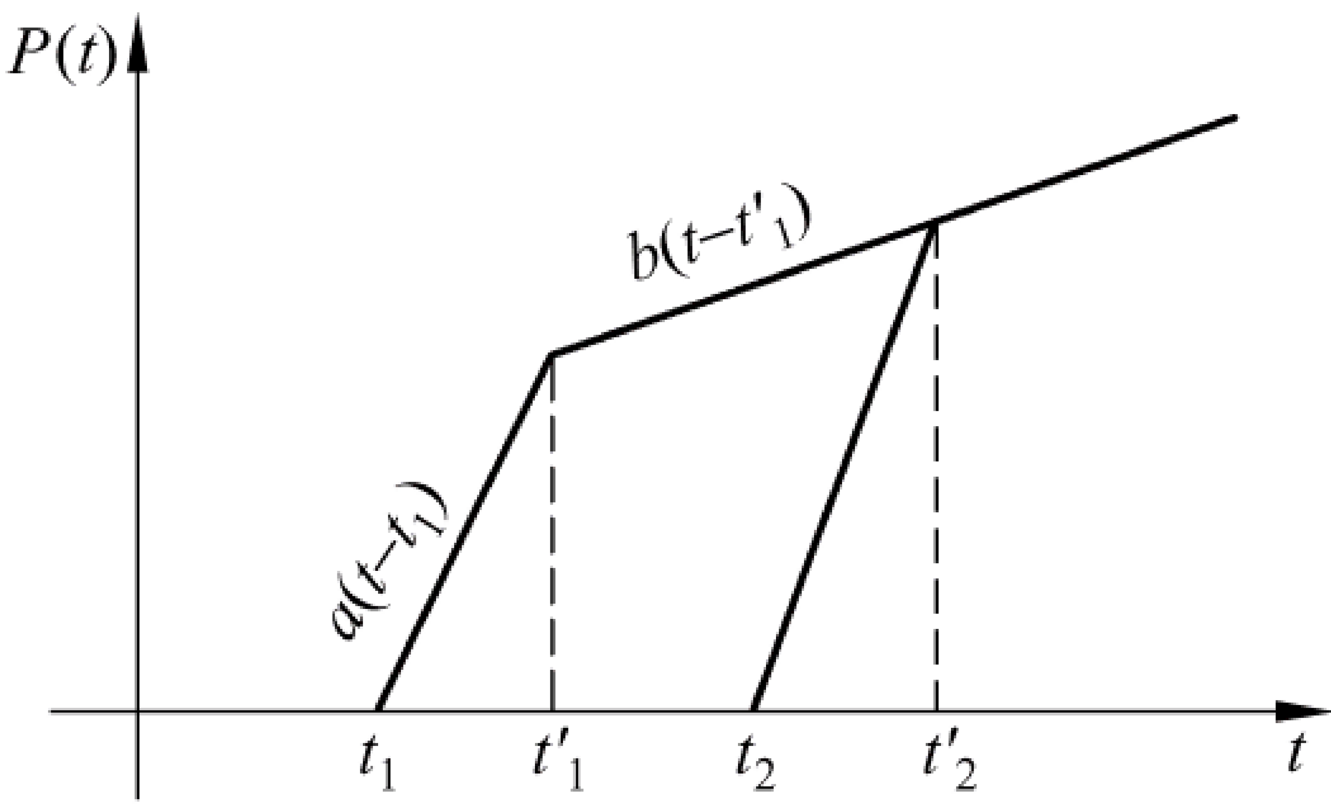



图 4.6　优先级变化曲线

另外,当享受服务进程队列为空时,新创建进程队列的头一个进程也将移入享受服务进程队列。

显然,在上述线性优先级调度法中,$a > b > 0$ 的条件是必要的。否则,当 $b > a > 0$ 时,两个不同队列中的就绪态进程的优先级将永远不会相等,从而,在享受服务进程队列中永远只有一个进程。因此,上述线性优先级调度策略退回到 FCFS 方式。另外,如果 $a > b = 0$,则线性优先级调度策略退回到轮转法调度方式。事实上,线性优先级调度策略是一种介于轮转法和 FCFS 方式之间的调度策略。这几种方式的调度性能,将在 4.5 节中更进一步讨论。

**5. 最短作业优先法(Shortest Job First,SJF)**

在批处理为主的系统中,如果采用 FCFS 方式进程作业调度,虽然系统开销小,算法简单,但是,如果估计执行时间很短的作业是在那些长作业的后面到达系统的话,则必须等待长作业执行完成之后才有机会获得执行。这将造成不必要的等待和某种不公平。最短作业优先法(SJF)就是选择那些估计需要执行时间最短的作业投入执行,为它们创建进程和分

配资源。直观上来说,采用最短作业优先的调度算法,可使得系统在同一时间内处理的作业个数最多,从而吞吐量也就大于其他调度方式。但是,对于一个不断有作业进入的批处理系统来说,最短作业优先法有可能使得那些长作业永远得不到调度执行的机会。

**6. 最高响应比优先法(Highest Response-ratio Next,HRN)**

最高响应比优先法是对 FCFS 方式和 SJF 方式的一种综合平衡。FCFS 方式只考虑每个作业的等待时间而未考虑执行时间的长短,而 SJF 方式只考虑执行时间而未考虑等待时间的长短,因此,这两种调度算法在某些极端情况下会带来某些不便。HRN 调度策略同时考虑每个作业的等待时间长短和估计需要的执行时间长短,从中选出响应比最高的作业投入执行。

响应比 $R$ 定义如下:

$$R = (W + T)/T = 1 + W/T$$

其中 $T$ 为该作业估计需要的执行时间,$W$ 为作业在后备状态队列中的等待时间。

每当要进行作业调度时,系统计算每个作业的响应比,选择其中 $R$ 最大者投入执行。这样,即使是长作业,随着它等待时间的增加,$W/T$ 也就随着增加,也就有机会获得调度执行。这种算法是介于 FCFS 和 SJF 之间的一种折中算法。由于长作业也有机会投入运行,在同一时间内处理的作业数显然要少于 SJF 法,从而采用 HRN 方式时其吞吐量将小于采用 SJF 法时的吞吐量。另外,由于每次调度前要计算响应比,系统开销也要相应增加。

## 4.5 算法评价

4.4 节中介绍了几种常用的作业和进程调度算法以及响应的调度性能衡量标准。本节主要利用解析技术从数学上分析几种主要调度方法的性能。

### 4.5.1 FCFS 方式的调度性能分析

设处理机或系统资源为服务器,一个进程或一个作业为享受该服务器服务的顾客。这些顾客按 FCFS 方式排队享受服务的系统模型如图 4.7 所示。

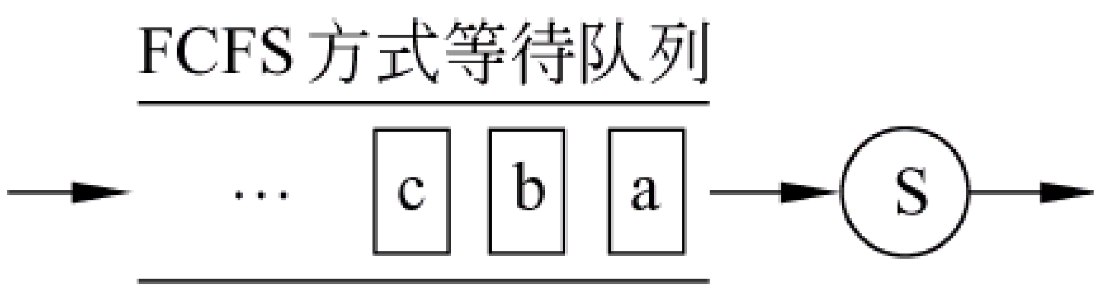


图 4.7 FCFS 方式的评价模型

这里,假定该系统模型中只有一个服务器 S。

设新顾客到达等待队列的时间与系统的当前状态、以前的顾客到达时间都无关,也就是新顾客到达系统的时间是服从泊松分布的。设 $\lambda$ 为到达率,则在单位时间内 $x$ 个顾客到达的概率为

$$P(x) = e^{-\lambda}\lambda^{x}/x!$$

单位时间内顾客到达的期望值,即算术平均值为

$$E(x) = \sum_{x=0}^{\infty} xP(x) = e^{-\lambda}\sum_{x=1}^{\infty}\lambda^{x}/(x-1)! = \lambda e^{-\lambda}\sum_{y=0}^{\infty}\lambda^{y}/y! \quad (y = x-1)$$

由于

$$\sum_{x=0}^{\infty}\lambda^{x}/x! = e^{\lambda}$$

所以,$E(x)=\lambda$。

也就是说，单位时间内顾客到达的平均值等于其到达率。

设服务器 $S$ 为顾客提供服务的概率也服从泊松分布，且 $\mu$ 为服务率，则单位时间内 $x$ 个顾客被服务的概率是

$$P(x) = e^{-\mu}\mu^{x}/x!$$

同理，单位时间内被服务顾客个数的算术平均值等于其服务率 $\mu$。

将单位时间换成任意时间 $t$，可得到在已知时间 $t$ 内 $x$ 个顾客到达的概率为

$$P(x(t)) = e^{\lambda t}(\lambda t)^{x}/x!$$

在 $t$ 时间内，一个顾客也不到达的概率为

$$P(0) = e^{-\lambda t}$$

从而，$t$ 时间内至少到达一个顾客的概率为

$$P(x(t) > 0) = 1 - e^{-\lambda t}$$

如果把时间 $t$ 换成固定的时间间隔 $\tau$，则有，在任何时间间隔 $\tau$ 内至少有一个到达发生的概率仍为 $1-e^{-\lambda t}$。这个概率和上一次顾客到达的时刻无关。新顾客在下一个 $\tau$ 时间内到达的概率和以前顾客的到达无关，这个特性称为无记忆特性或马尔可夫性质。利用该性质，可以简化顾客和服务的排队模型。

由于服务器的服务概率也服从泊松分布，也可推出它在 $\tau$ 时间间隔内至少为一个顾客服务的概率为 $1-e-\mu t$，且与以前的服务过程无关，因此，服务器的服务特性也是满足马尔可夫性质的。

由于

$$P(x(t) > 0) = 1 - e^{-\lambda t}$$

其密度函数为

$$P(x(t) > 0) = \lambda e^{-\lambda t}$$

$t$ 的期望值等于

$$E(t)\int_0^{\infty} t\lambda e^{-\lambda t}\,dt = - te^{-\lambda t}\ \Big|_0^{\infty} + \int_0^{\infty} e^{-\lambda t} = 1/\lambda$$

即两个连续到达的顾客之间的平均时间间隔为 $1/\lambda$。同理，可得服务器服务时间的平均值为 $1/\mu$。显然，只有当 $1/\mu<1/\lambda$，也就是 $\lambda<\mu$ 时，系统才是稳定的。否则，等待服务队列将无限增长。

设 $S_i$ 为系统的一个状态，表示等待服务的等待队列中有 $i-1$ 个顾客，服务器中有 1 个顾客存在。由概率密度函数可知，在 $dt$ 时间内 1 个新顾客到达的概率是

$$P(dt\text{ 时间内 1 个顾客到达}) = \lambda e^{-\lambda dt}\,dt$$

将上式做多项式展开得

$$\lambda e^{-\lambda dt}\,dt = \lambda dt + O(dt^2)$$

同理可得，在 $dt$ 时间内服务器为 1 个顾客提供服务的概率是

$$P(dt\text{ 时间内 1 个顾客离开}) = \mu dt + O(dt^2)$$

省略以上两式的二次方项，在 $dt$ 时间内 1 个顾客到达或离开的概率为

$$P(dt\text{ 时间内 1 个顾客到达}) = \lambda dt$$

$$P(dt\text{ 时间内 1 个顾客离开}) = \mu dt$$

对于 $i=0$ 时，有

$$P(\mathrm{d}t\text{ 时间内 1 个顾客离开}) = 0$$

在 d$t$ 时间内，顾客一个也不到达和顾客一个也不离开的概率为

$$1 - P(\mathrm{d}t\text{ 时间内 1 个顾客到达}) - P(\mathrm{d}t\text{ 时间内 1 个顾客离开})$$
$$- P(\mathrm{d}t\text{ 时间内 2 个以上顾客到达}) - P(\mathrm{d}t\text{ 时间内 2 个以上顾客离开})$$

由于上式的后两项实际上是 d$t$ 的二次方以上的函数，从而上式可合并为

$$P(\mathrm{d}t\text{ 时间内不发生变化}) = 1 - (\lambda + \mu)\,\mathrm{d}t - O(\mathrm{d}t^2) \quad (i > 0)$$

或

$$P(\mathrm{d}t\text{ 时间内不发生变化}) = 1 - \lambda\mathrm{d}t - O(\mathrm{d}t^2) \quad (i = 0)$$

忽略二次方项，可得状态变化图如图 4.8 所示。

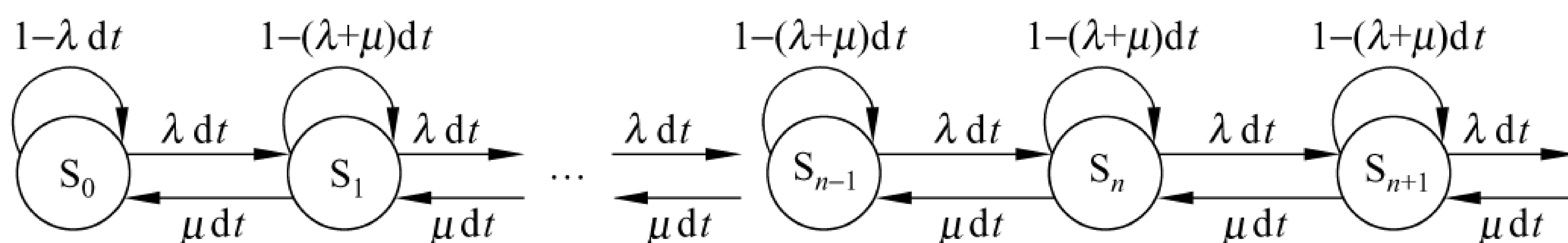



图 4.8　状态转换图

设系统在 $t+\mathrm{d}t$ 时间内处于状态 $S_i$ 的概率为 $P_i(t+\mathrm{d}t)$，则由状态转换图有

$$P_i(t + \mathrm{d}t) = (1 - (\lambda + \mu)\mathrm{d}t)P_i(t) + \lambda\mathrm{d}t\, P_{i-1}(t) + \mu\mathrm{d}t\, P_{i+1}(t)$$

式中 $i \geqslant 1$。对于 $i=0$ 时，有

$$P_0(t + \mathrm{d}t) = (1 - \lambda\mathrm{d}t)P_0(t) + \mu\mathrm{d}t\, P_1(t)$$

当系统到达稳定状态时，上述概率将会趋于一个常量，$P_i(t+\mathrm{d}t)$的导数为 0，即

$$P_i'(t + \mathrm{d}t) = -(\lambda + \mu)P_i(t) + \lambda\, P_{i-1}(t) + \mu\, P_{i+1}(t) = 0$$
$$P_0'(t + \mathrm{d}t) = -\lambda\, P_0(t) + \mu\, P_i(t) = 0$$

即

$$P_1 = (\lambda/\mu)\; P_0(t)$$
$$(\lambda + \mu)\; P_i(t) = \lambda\, P_{i-1}(t) + \mu\, P_{i+1}(t)$$

令 $\lambda/\mu=\rho$，采用代入法可得

$$P_i(t) = \rho^i P_0(t)$$

另外，系统运行中的任一时刻总是处于图 4.8 所示状态图中的任一状态中，从而有

$$\sum_{i=0}^{\infty} P_i = 1$$

由此可得

$$\sum_{i=0}^{\infty} \rho^i P_0(t) = 1$$

由于在 $\rho<1$ 时有

$$\sum_{i=0}^{\infty} \rho^i = 1/(1 - \rho)$$

从而有

$$P_0(t) = 1 - \rho$$

即，在稳态条件下，系统内不存在顾客的概率为 $1-\rho=(\mu-\lambda)/\mu$，而系统内存在顾客的概率为 $\lambda/\mu$。系统内顾客的算术平均值是

$$n = E(i) = \sum_{i=0}^{\infty} iP_i(t) = \sum_{i=0}^{\infty}(1-\rho)^i\rho^i = \rho(1-\rho)$$

把上述按 FCFS 方式排列和调度，并只有一个服务器的系统称为 M/M/1 系统。第一个字母 M 表示顾客到达时间间隔是指数分布，具有马尔可夫性质；第二个字母 M 表示从服务器离开的顾客的时间间隔服从指数分布，具有马尔可夫性质；数字 1 表示只有一个服务器。

设响应时间 $R$ 为从顾客到达等待队列后开始到离开服务器的时间，则在系统进入稳定状态之后，系统中的顾客数 $n$ 和平均响应时间之间存在一个非常简单的关系，即 $n=\lambda R$，$\lambda$ 为顾客的平均到达率。该公式称为 Little 结果(Little's result)，是一个在系统分析中广泛使用的公式。

利用 Little 结果，可以求出 M/M/1 系统的平均响应时间：

$$R = n/\lambda = \rho/\lambda(1/(1-\rho)) = 1/(\mu(1-\rho))$$

平均响应时间和 $\rho$ 的关系图如图 4.9 所示。

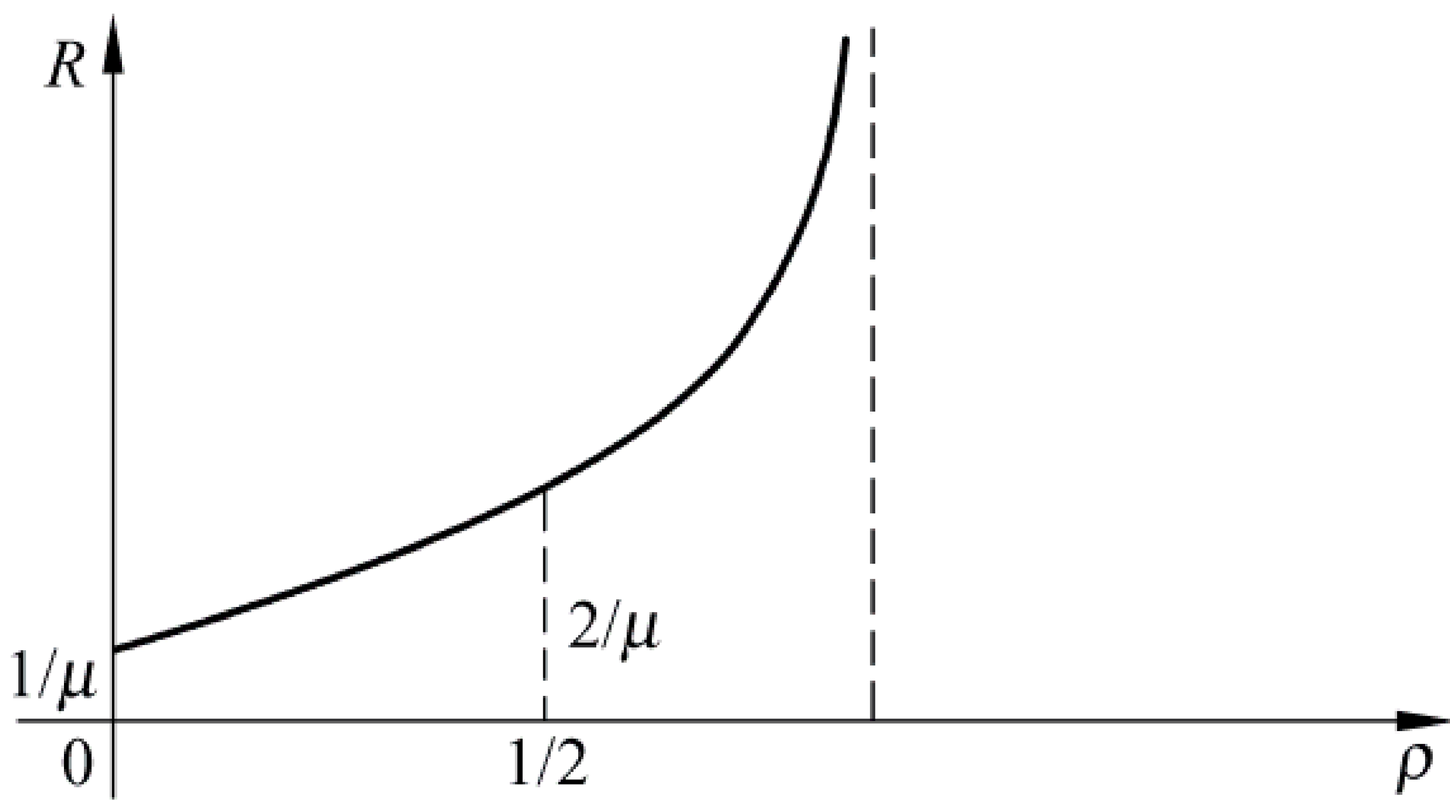



图 4.9 平均响应时间 $R$ 和 $\rho$ 的关系

由图 4.9 可以看出，M/M/1 系统的服务性能是由 $\rho$ 决定的。如果 $\rho$ 趋近 1，即 $\lambda$ 趋近 $\mu$，则响应时间急剧增大，系统性能变差；而当 $\rho$ 小于 1/2 时，等待队列中为空的可能性较大，因为平均响应时间小于平均服务时间的 2 倍。

下面再来看看 M/M/1 系统对短作业或短进程的影响。

设短作业的到达率和服务率分别为$(\lambda_1,\mu_1)$，长作业的到达率和服务率为$(\lambda_2,\mu_2)$，且二者都服从泊松分布。二者的合成仍是泊松过程(请读者自己证明)，其到达率 $\lambda=\lambda_1+\lambda_2$，平均服务时间为

$$\begin{aligned} 1/\mu &= \lambda_1/(\lambda_1+\lambda_2) * 1/\mu_1 + \lambda_2/(\lambda_1+\lambda_2) * 1/\mu_2 \\ &= 1/(\lambda_1+\lambda_2) * (\lambda_1/\mu_1 + \lambda_2/\mu^2) \end{aligned}$$

从而有

$$\lambda/\mu = \lambda_1/\mu_1 + \lambda_2/\mu_2 = \rho_1 + \rho_2$$

一般来说，短作业的服务时间 $1/\mu_1$ 远小于长作业的服务时间 $1/\mu_2$。FCFS 调度策略时有响应时间 $R$ 为

$$R = 1/\lambda * \rho/(1-\rho)$$

其中，$\lambda=\lambda_1+\lambda_2$，$\rho=\rho_1+\rho_2$。

由于 $\lambda$ 和 $\rho$ 中包含了 $\lambda_1$、$\lambda_2$、$\mu_1$ 和 $\mu_2$，所有作业的平均响应时间相同，从而，短作业在系统中的驻留平均时间与长作业的驻留平均时间相同，这对短作业是不利的。

### 4.5.2 轮转法调度性能评价

轮转法调度时的顾客到达率要大大高于 FCFS 方式。设时间片为 $q$，服务时间平均值为 $1/\mu$，每个顾客平均需要 $k$ 个时间片。从而有 $1/\mu=k*q$。如果某个顾客刚好需要 $k$ 个时间片的服务时间，则这个顾客就会到达等待队列 $k$ 次($k>1$)。

对于短作业，$1/\mu_1=k_1*q$，而对于长作业，$1/\mu_2=k_2*q$。设新顾客的到达率 $\lambda=\lambda_1+\lambda_2$，服务时间为

$$1/\mu = (\lambda_1/\mu)\ k_1 * q + (\lambda_2/\mu)\ k_2 * q = (1/\lambda)(\lambda_1/\mu_1 + \lambda_2/\mu_2)$$

从而有

$$\lambda/\mu = \rho = \rho_1 + \rho_2$$

这看上去好像在平均响应时间方面，轮转法和 FCFS 调度方式差别不大。但事实上，在等待队列中享受过 $k$ 次时间片服务的顾客的响应时间是

$$R(k) = \rho/(\lambda(1-\rho)) = 1/(\mu(1-\rho)) = k * q\ /(1-\rho)$$

也就是说，响应时间与服务时间成正比。从而，所需服务时间短的顾客的响应时间将会小于所需服务时间长的顾客的响应时间。因此，轮转法在响应时间上要优于 FCFS 调度方式。

### 4.5.3 线性优先级法的调度性能

线性优先级调度策略（SRR）是介于 FCFS 方式和轮转法之间的一种调度策略。SRR 方式把新到达的顾客首先送入等待室休息一段时间后，再送到等待服务队列（见图 4.10）。设顾客到达等待室的到达率为 $\lambda$，到达等待队列的到达率为 $\lambda'$。$\lambda'$依赖于等待室的到达率 $\lambda$ 和线性参数 $r$，其中 $r=1-b/a$。由 4.4 节可知，有 $a>b$，且 $a$ 和 $b$ 分别为等待室内顾客和等待队列中顾客优先级的线性增加系数。

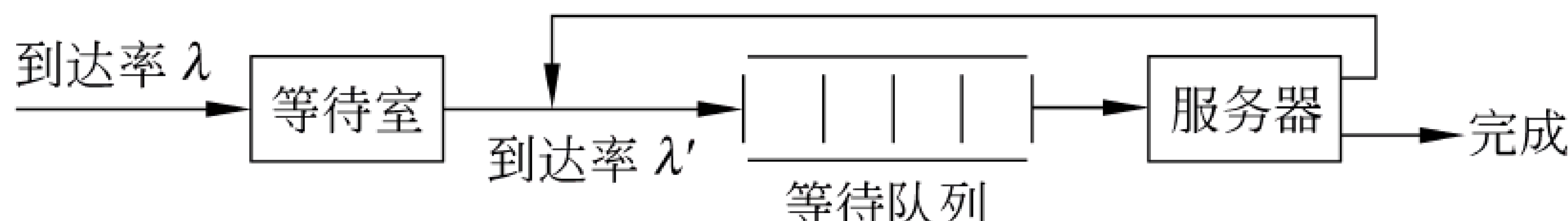



图 4.10　线性优先级调度的评价模型

设 $t_1$ 和 $t_2$ 分别为两个顾客接连到达等待室的时间，则有

$$1/\lambda = t_2 - t_1$$

设 $t_1'$和 $t_2'$分别为两个顾客从等待室接连到达等待队列的时间，则有

$$1/\lambda' = t_2' - t_1'$$

由图 4.6 可知，有

$$(t_2 - t_1)/(t_2' - t_1') = r$$

即

$$\lambda'/\lambda = r,\quad \lambda' = r\lambda$$

由于 $r<1$，所以等待室以 $r$ 的比率滞留到达的顾客。

线性优先级调度系统的响应时间是 $R_{sr}$。且 $R_{sr}$ 为等待室的平均等待时间 $R_d$ 和进入等待队列后得到服务的平均响应时间 $R_s$ 之和。由对轮转法的分析，则有：

$$R_s = 1/(\mu - \lambda)$$

$$r * R_s = 1/(\mu - \lambda')$$

从而

$$R_d = r * R_s - R_s = 1/(\mu - \lambda') - 1/(\mu - \lambda) = (\lambda' - \lambda)\ /(\mu - \lambda)(\mu - \lambda')$$

另外，由轮转法可知：

$$R_s(k) = kq/(1-\rho)$$

其中，$\rho=\lambda/\mu$。从而有

$$R_{sr}(k) = R_d + R_s(k) = (\lambda' - \lambda)/(\mu - \lambda)(\mu - \lambda') + kq/(1-\rho)$$

$$=1/(\mu-\lambda')-(1-kq\mu)/(\mu-\lambda)$$

其中 $k$ 是一个顾客享受服务的时间片的次数，$q$ 是时间片常数。

可以比较一下 FCFS 方式、SRR 方式以及轮转法 3 种调度方式的平均响应时间。

采用 FCFS 方式时，有：

$$R_{fc}(k)=1/(\mu-\lambda)$$

采用轮转法时有

$$R_{rr}(k)=kq\mu/(\mu-\lambda)$$

采用 SRR 方式时，则响应时间为

$$R_{sr}(k)=1/(\mu-\lambda')-(1-kq\mu)/(\mu-\lambda)$$

如果 $kq=1/\mu$，即顾客的服务时间与其平均服务时间相等的话，则 $R_{sr}(k)$ 中第二项变为 0，上述 $R_{fc}(k)=R_{rr}(k)=R_{sr}(k)$。当然，对于需要服务时间短的顾客来说，有 $kq<1/\mu$，而对于需要服务时间长的顾客来说，则有 $kq>1/\mu$。从而，对于服务时间短的顾客，其响应时间为

$$R_{rr}<R_{sr}<R_{fc}$$

而对于服务时间长的顾客来说，其响应时间则为

$$R_{fc}<R_{sr}<R_{rr}$$

上面只是从响应时间的角度对几种常见的调度策略进行了评价分析。除了响应时间之外，CPU 利用率也是评价调度性能的另一个标准。

# 4.6 实时系统调度方法

## 4.6.1 实时系统的特点

随着移动通信和网络计算技术的发展，实时系统正变得越来越重要。操作系统是实时系统中最重要的部分之一，它负责在用户要求的时限内进行事件处理和控制。

实时系统与其他系统最大的区别在于，其处理和控制的正确性不仅仅取决于计算的逻辑结果，而且取决于计算和处理结果产生的时间。因此，实时系统的调度与工业生产中的生产过程调度有许多相同之处，即把给定的任务按所要求的时限调配到相应的设备上去处理完成。

根据对处理外部事件的时限(deadline)要求，实时系统中处理的外部事件可分为硬实时任务(hard real time task)和软实时任务(soft real time task)。硬实时任务要求系统必须完全满足任务的时限要求；软实时任务则允许系统对任务的时限要求有一定的延迟，其时限要求只是一个相对条件。

实时系统的另一个特点是它所处理的外部任务可分为周期性的与非周期性的两大类。对于非周期性任务来说，必定存在一个完成或开始进行处理的时限，而周期性任务只要求在周期 $T$ 内完成或开始进行处理。

一般来说，实时操作系统具有以下特点：

(1) 有限等待时间(决定性特性)。

(2) 有限响应时间。

（3）用户控制。

（4）可靠性高。

（5）系统出错处理能力强。

分时系统中并发执行的进程具有不确定性，其执行顺序与执行环境有关。实时系统则不然，它要求所有的进程在处理事件时都必须在有限时间内开始，这一特性又被称为实时系统的决定性特性。

实时系统的有限响应时间特性是指从系统响应外部事件开始，必须在有限时间内处理完毕。

另外，在分时的非实时系统中，用户不能参与对进程调度的控制。在实时系统中，用户可以控制进程的优先级并选择相应的调度算法，从而达到对进程执行先后顺序的控制。

实时系统要求很高的可靠性。在分时系统的非实时系统中，用户可以用重新启动计算机等措施来处理系统出错。但是，实时系统主要是对外部事件进行处理和控制，例如导弹系统的控制，这样的系统不允许出现控制错误。

另外，当系统发生错误时，实时系统不能像非实时系统那样，先停止当前处理的用户程序，转去执行出错处理或使系统自动退出。实时系统要求系统在出错时，既能够处理所发生的错误，又不影响当前正在执行的用户应用。

实时系统的上述特性要求实时操作系统具有下述能力：

（1）很快的进程或线程切换速度。

进程或线程切换速度是实时系统设计的核心。与分时系统不同，公平性以及最小平均响应时间等指标在实时系统中并不重要，实时系统中调度算法的设计原则是满足所有硬实时任务的处理时限和尽可能多地满足软实时任务的处理时限。

（2）快速的外部中断响应能力。

有关中断处理和响应的详细介绍在第 9 章中给出，不过，只有对外部中断信号反应迅速，系统才能对外部事件作出迅速反应。

（3）基于优先级的随时抢先式调度策略。

基于优先级的调度策略大致有以下 4 种：

① 优先级＋时间片轮转调度策略；

② 基于优先级的非抢先式调度策略；

③ 基于优先级的固定点抢先式调度策略；

④ 基于优先级的随时抢先式调度策略。

对于调度策略①来说，因为调度必须在时间片到来时才能发生，实时进程必须等待占有处理机的进程执行到时间片结束时才能获得处理机，因此，这种方法不能用作实时调度。同理，基于优先级的非抢先式调度策略也不能用作实时调度，因为高优先级的实时进程只有在当前执行进程自动让出处理机之后才能获得处理机。

基于优先级的固定点抢先式调度方式与基于优先级的随时抢先式调度策略是实时系统的主要调度策略。基于优先级的固定点抢先式调度方式与优先级＋时间片轮转调度方式有相似之处，其主要区别在于允许抢先的固定点间隔要比时间片小得多，并保证能满足所有硬实时的处理时限。

### 4.6.2 实时调度算法的分类

实时调度算法分为如下 4 类：

**1. 静态表格驱动类**

静态表格驱动类的实时调度算法，对可能的调度条件和参数进行静态分析，并将分析结果作为实际调度结果。这类调度算法多用于调度处理周期性任务，其主要分析参数为周期，执行时间、周期执行结束时限和任务优先级等。最早时限优先法是比较典型的静态表格驱动算法。这里，最早时限优先法是优先调度时限最早的任务获得处理机的调度算法。

**2. 静态优先级驱动抢先式调度算法类**

该类算法也进行静态分析，不过，它们的静态分析不直接产生调度结果，而只用来指定任务的优先级。频率单调调度算法就是一种静态优先级驱动的抢先式调度算法。

**3. 动态计划调度算法类**

动态计划调度算法在调度任务执行之前排出调度计划，并分析计划的调度结果是否使得任务所要求的处理时限得到满足。如果能够满足，则按调度计划执行，否则修改调度计划。

**4. 尽力而为调度算法类**

这一类算法不进行可能性分析，只对到达的事件和相关任务指定相应的优先级，并进行调度。尽力而为调度方式开销较小，实现容易。但是，该算法不一定满足用户要求的处理时限。

### 4.6.3 时限调度算法与频率单调调度算法

时限调度算法是一种以满足用户要求的时限为调度原则的算法。在实时系统中的用户要求时限有两种，即处理开始时限（starting deadline）和处理结束时限（ending deadline）。时限调度算法可以使用任一种时限。时限调度算法可用于周期性调度与非周期性调度两种。

时限调度算法所需要的相关输入信息包括以下几种：

**1. 任务就绪时间或事件到达时间**

任务就绪时间或事件到达时间指的是进程进入就绪状态，可以被调度执行的时间。对于周期性任务来说，该时间是可以预知的，而且时间间隔是周期性的。对于非周期性任务来说，这些时间可能是可预知的，但大部分时候是不可预知的，需要事件发生来驱动。

**2. 开始时限**

开始时限指处理机必须开始对任务进行处理的时限。

**3. 完成时限**

完成时限指任务必须完成的时间。

**4. 处理时间**

处理时间指完成相关任务所需占用处理机的时间。

**5. 资源需求**

资源需求指除了处理机之外，另外还需要的其他硬软件资源。如果所处理的任务有处理机之外的其他资源需求，则调度算法要复杂得多。

**6. 优先级**

优先级可由分析计算后获得,也可根据时限要求由用户指定。

时限调度算法的基本思想是:按用户的时限要求顺序设置优先级,优先级高者占据处理机,也就是说,时限要求最近的任务优先占有处理机。

时限调度是抢先式的。抢先式时限调度算法必须把新到达任务的时限要求和当前正在执行任务的时限要求进行比较,如果新到达任务的时限要求更近,则应执行新到达的任务。

下面是一个使用时限调度算法调度周期性实时任务的例子。

设实时系统从两个不同的数据源 DA 和 DB 周期性地收集数据并进行处理,其中 DA 的时限要求以 30ms 为周期,DB 的时限要求以 75ms 为周期。设 DA 所需处理时限为 15ms,DB 所需处理时限为 38ms,则与 DA 和 DB 有关进程的事件发生时限(就绪时限),执行时限以及结束时限如表 4.1 所示。

**表 4.1 周期性任务的预计发生、执行与结束时限**

| 进程 | 事件发生时限/ms | 执行时限/ms | 结束时限/ms |
|---|---|---|---|
| DA(1) | 0 | 15 | 30 |
| DA(2) | 30 | 15 | 60 |
| DA(3) | 60 | 15 | 90 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| DB(1) | 0 | 38 | 75 |
| DB(2) | 75 | 38 | 150 |
| DB(3) | 150 | 38 | 225 |
| ⋮ | ⋮ | ⋮ | ⋮ |

如果使用时限调度算法,并按最近结束时限优先级最高的方法进行排列,可以给出表 4.1 所示各进程的调度顺序和相对时间(见图 4.11)。

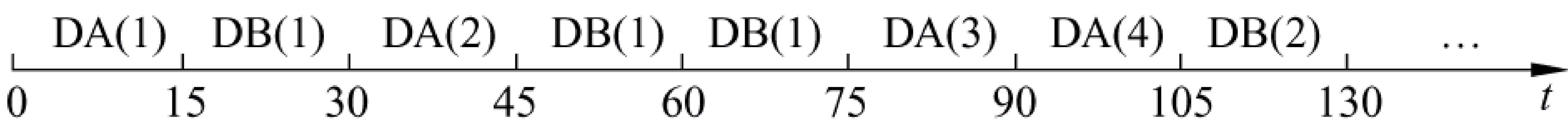



图 4.11 时限调度算法给出的调度顺序

如图 4.11 所示,在开始时,进程 DA(1)和 DB(1)的结束时限比较,DA(1)的结束时限最近,从而调度进程 DA(1)执行。DA(1)的实际结束时间为 15ms,小于 30ms 的时限要求。紧接着,进程 DB(1)被调度执行。在执行到时间为 30ms 时,进程 DA(2)进入就绪状态。由于 DA(2)的结束时限为 60ms,近于 DB(1)的结束时限 75ms,从而 DB(1)被 DA(2)抢先。DA(2)的实际结束时间为 45ms,小于要求时限 60ms。

在 DA(2)结束之后,DB(1)再次占有处理机继续执行,当 DB(1)执行到时间为 60ms 时,进程 DA(3)进入就绪状态。但是,由于 DA(3)的结束时限为 90ms,远于 DB(1)的结束时限 75ms,从而 DB(1)继续执行。

时限调度算法也可以用于非周期性任务调度。

频率单调调度算法是一种被广泛用于多周期性实时处理的调度算法。其基本原理是频率越低(周期越长)的任务的优先级越低。

另外,设周期性任务的执行时间为 $C$,则使用频率单调调度算法的必要条件是 $C \leqslant T$。

已经证明，对于 $n(n\geqslant 1)$ 个周期的不同任务来说，设每个周期为 $T_i$，其相应任务的执行时间为 $C_i$，则使用频率单调调度算法的充分条件是

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_i}{T_i}+\cdots+\frac{C_n}{T_n}\leqslant n(2^{\frac{1}{n}}-1)$$

例如，对于由 3 个周期组成的实时任务序列来说，其执行时间与周期之比应是

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\frac{C_3}{T_3}\leqslant 3(2^{\frac{1}{3}}-1)=0.799$$

如果进程执行时间与周期比之和大于 $n(2^{1/n}-1)$，则用户所要求的时限无法保证。

## 本 章 小 结

CPU 是计算机系统中十分重要的资源，本章主要介绍处理机的调度目标、策略以及评价方法等。

因为处理机调度程序不可能选择全部驻留在外存的进程，因此，在调度一个进程占有处理机之前，系统必须按某种策略把外存中处于后备状态的作业选择出来，并创建进程和分配内存，为进程执行准备必需的资源。这一步称为作业调度或高级调度。作业调度的目标是尽量做到公平合理，能执行尽可能多的作业、尽可能快的响应时间以及高的设备利用率等。任一调度算法要同时满足这些调度目标是不可能的。大多数操作系统都是根据用户需要而采用兼顾某些目标的方法。比较常用的作业调度算法有 FCFS 方法、SJF(最短作业优先)法和 HRN(最高响应比)法等。这几种方法各有特点。其中 FCFS 法系统开销小，且对每个作业来说按其到达顺序被依次调度。FCFS 法不利于短作业。SJF 法可得到最大系统吞吐率，即每天处理的作业个数最多。但是 SJF 法有可能使长作业永远没有机会执行。HRN 法是介于 FCFS 法和 SJF 法之间的一种方法。

除了作业调度之外，还介绍了一种称为交换调度的中级调度。在有的系统中，把那些处于等待状态或就绪状态的进程换出内存，而把那些等待事件已经发生或已在外存交换区中等待了较长时间的进程换入内存。

只有在进程被建立起来并且已获得足够的资源之后，系统才使用进程调度策略把处理机分配给选出进程。因此，处理机的调度涉及 3 个层次的调度。进程调度的主要任务是选择一个合适的进程占据处理机。系统的要求不一样，进程调度方法变化较大。比较常用的有 RR(轮转)法、FCFS(先来先服务)法、优先级法和 SRR(线性优先级)法等。其中轮转法主要用于分时系统，它具有较好的响应时间，且对每个进程来说都具有较好的公平性。FCFS 法不利于执行时间短的进程，而 SRR 法则是介于 FCFS 法和 RR 法之间的一种进程调度方法。

## 习　　题

4.1　什么是分级调度？分时系统中有作业调度的概念吗？如果没有，为什么？

4.2　试述作业调度的主要功能。

4.3　作业调度的性能评价标准有哪些？这些性能评价标准在任何情况下都能反映调度策

略的优劣吗？

4.4 进程调度的功能有哪些？

4.5 进程调度的时机有哪几种？

4.6 假设有 4 道作业，它们的提交时刻及执行时间由下表给出：

| 作业号 | 提交时刻/hh:mm | 执行时间/hr |
| --- | --- | --- |
| 1 | 10:00 | 2 |
| 2 | 10:20 | 1 |
| 3 | 10:40 | 0.5 |
| 4 | 10:50 | 0.3 |

计算在单道程序环境下，采用先来先服务调度算法和最短作业优先调度算法时的平均周转时间和平均带权周转时间，并指出它们的调度顺序。

4.7 设某进程所需要的服务时间 $t=k*q$，其中，$k$ 为时间片的个数，$q$ 为时间片长度且为常数。当 $t$ 为一定值时，令 $q$ 趋于 0，则有 $k$ 趋于无穷，从而服务时间为 $t$ 的进程响应时间 $T$ 为 $t$ 的连续函数。对应于时间片的轮转调度方式(RR)、先来先服务方式(FCFS)和线性优先级调度方式(SRR)，其响应时间函数分别为

$$T_{rr}(t) = t*\mu/(\mu-\lambda)$$

$$T_{fc}(t) = 1/(\mu-\lambda)$$

$$T_{sr}(t) = 1/(\mu-\lambda)-(1-t*\mu)/(\mu-\lambda')$$

其中，$\lambda'=((1-b)/a)*\lambda=r*\lambda$。

取$(\lambda,\mu)=(50,100)$和$(\lambda,\mu)=(80,100)$，分别改变 $r$ 的值，画出 $T_{rr}(t)$、$T_{fc}(t)$和 $T_{sr}(t)$的时间变化图。

4.8 什么是多处理机系统？并行处理系统、计算机网络、分布式系统和多处理机系统的操作系统之间有何区别？

4.9 什么是实时调度？它与非实时调度有何区别？

4.10 写出表 4.1 所示周期性任务调度用的时限调度算法。

4.11 设周期性任务 $P_1,P_2,P_3$ 的周期 $T_1,T_2,T_3$ 分别为 100，150，350，执行时间分别为 20，40，100，问：是否可用频率单调调度算法进行调度？

# 第5章　存储管理

## 5.1　存储管理的功能

存储器是计算机系统的重要资源之一。因为任何程序和数据以及各种控制用的数据结构都必须占用一定的存储空间，因此，存储管理直接影响系统性能。

存储器由内存(primary storage)和外存(secondary storage)组成。内存由顺序编址的块组成，每块包含相应的物理单元。CPU要通过启动相应的输入输出设备后才能使外存与内存交换信息。本章主要讨论内存管理问题，内容包括几种常用的内存管理方法、内存的分配和释放算法、虚拟存储器的概念、控制主存和外存之间的数据流动方法、地址变换技术和内存数据保护与共享技术等。下面先介绍存储管理的功能。

### 5.1.1　虚拟存储器

虚拟存储器是存储管理的核心概念。现代计算机系统的物理存储器都分为内存和外存，其理由是内存价格昂贵，不可能用大容量的内存存储所有被访问的或不被访问的程序与数据段。而外存尽管访问速度较慢，但价格便宜，适合存放大量信息。实验证明，在一个进程的执行过程中，其大部分程序和数据并不经常被访问。这样，存储管理系统把进程中那些不经常被访问的程序段和数据放入外存中，待需要访问它们时再将它们调入内存。那么，对于那些一部分数据和程序段在内存而另一部分在外存的进程，怎样安排它们的地址呢?

通常由用户编写的源程序，首先要由编译程序编译成CPU可执行的目标代码。然后，链接程序把一个进程的不同程序段链接起来以完成所要求的功能。显然，对于不同的程序段，应具有不同的地址。有两种方法安排这些编译后的目标代码的地址。一种方法是按照物理存储器中的位置赋予实际物理地址。这种方法的好处是CPU执行目标代码时的执行速度高。但是，由于物理存储器的容量限制，能装入内存并发执行的进程数将会大大减少，对于某些较大的进程来说，当其所要求的总内存容量超过内存容量时将会无法执行。另外，由于编译程序必须知道内存的当前空闲部分及其地址，并且把一个进程的不同程序段连续地存放起来，因此编译程序将非常复杂。

另一种方法是编译链接程序把用户源程序编译后链接到一个以0地址为始地址的线性或多维虚拟地址空间。这里，链接既可以是在程序执行以前由链接程序完成的静态链接，也可以是在程序执行过程中由于需要而进行的动态链接。而且，每一个进程都拥有这样一个空间(这个空间是一维的还是多维的由存储管理方式决定)。每个指令或数据单元都在这个虚拟空间中拥有确定的地址，这个地址称为虚拟地址(virtual address)。显然，进程在该空间的地址排列可以是非连续的，其实际物理地址由虚拟地址到实际物理地址的变换得到。由源程序到实际存放该程序指令或数据的内存物理位置的变换如图5.1所示。

进程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟存储器(virtual store

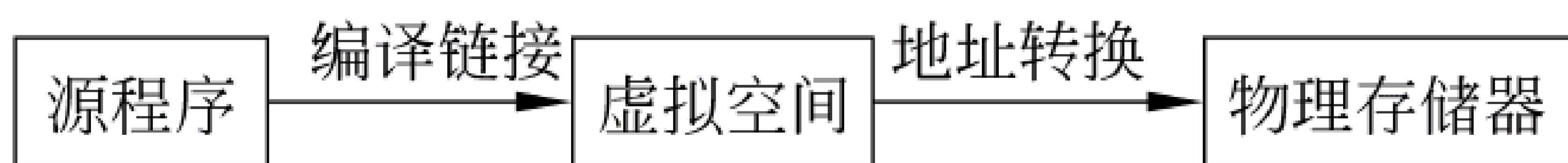



图 5.1　地址变换与物理存储器

或 virtual memory)。虚拟存储器不考虑物理存储器的大小和信息存放的实际位置,只规定每个进程中互相关连的信息的相对位置。与实际物理存储器数量有限,且被所有进程共享不一样,每个进程都拥有自己的虚拟存储器,且虚拟存储器的容量是由计算机的地址结构和寻址方式确定的。例如,直接寻址时,如果 CPU 的有效地址长度为 16 位,则其寻址范围为 0～64K。

图 5.1 中的编译和链接主要是语言系统的设计问题,而虚拟存储器到物理存储器的变换是操作系统必须解决的问题。要实现这个变换,必须要有相应的硬件支持,并使这些硬件能够完成统一管理内存和外存之间数据和程序段自动交换的虚拟存储器功能。即,由于每个进程都拥有自己的虚存,且每个虚存的大小不受实际物理存储器的限制,因此,系统不可能提供足够大的内存来存放所有进程的内容。内存中只能存放那些经常被访问的程序和数据段等。这就需要有相当大的外部存储器,以存储那些不经常被访问或在某一段时间内不会被访问的信息。待到进程执行过程中需要这些信息时,再从外存中自动调入内存。至于如何具体实现和管理虚拟存储器,将在后面有关章节介绍。

## 5.1.2　地址变换

内存地址的集合称为内存空间或物理地址空间。内存中,每一个存储单元都与相应的称为内存地址的编号相对应。显然,内存空间是一维线性空间。

怎样把几个虚存的一维线性空间或多维线性空间变换到内存的唯一的一维物理线性空间呢? 这涉及两个问题。

第一个问题是虚拟空间的划分问题。例如进程的正文段和数据段应该放置在虚拟空间的什么地方。虚拟空间的划分使得编译链接程序可以把不同的程序模块(它们可能是用不同的高级语言编写的)链接到一个统一的虚拟空间中。虚拟空间的划分与计算机系统结构有关,例如,VAX-11 型机中的虚拟空间划分为进程空间和系统空间两大部分,而进程空间又更进一步划分为程序区和控制区。VAX-11 的虚拟空间容量为 $2^{32}$ 个单元,其中程序区占 $2^{30}$ 个单元,用来存放用户程序,程序段以零为基址动态地向高地址方向增长,最大可达 $2^{30}-1$ 号单元。控制区也占 $2^{30}$ 个单元,存放各种方式和状态下的堆栈结构及数据等,其虚拟地址由 $2^{31}-1$ 号地址开始由高向低地址方向增长。系统空间占 $2^{31}$ 个单元,用来存放操作系统程序。VAX-11 机的虚拟空间结构如图 5.2 所示。

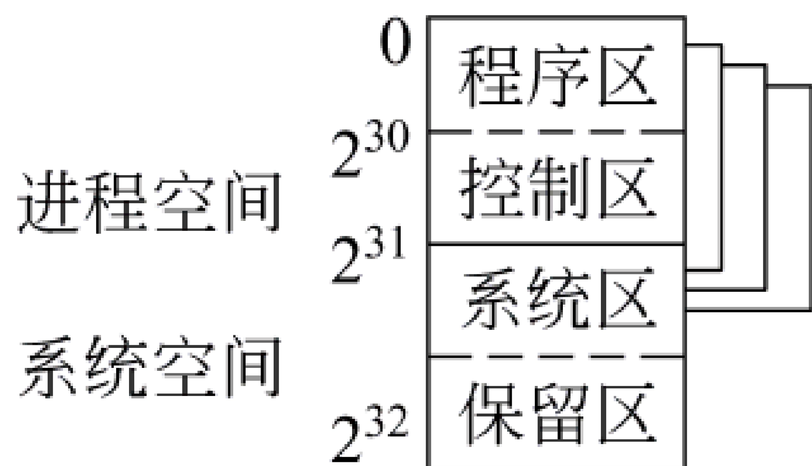



图 5.2　虚拟空间的划分

第二个问题是把虚拟空间中已链接和划分好的内容装入内存,并将虚拟地址映射为内存地址的问题,称之为地址重定位或地址映射。地址映射就是要建立虚拟地址与内存地址的关系。实现地址重定位或地址映射的方法有两种:静态地址重定位和动态地址重定位。

**1. 静态地址重定位**

静态地址重定位(static address relocation)是在虚拟空间程序执行之前由装配程序完

成地址映射工作。假定分配程序已分配了一块首地址为 BA 的内存区给虚拟空间内的程序段，且每条指令或数据的虚拟地址为 VA，那么，该指令或数据对应的内存地址为 MA，从而完成程序中所有地址部分的修改，以保证 CPU 的正确执行。显然，对于虚拟空间内的指令或数据来说，静态地址重定位只完成一个首地址不同的连续地址变换。它要求所有待执行的程序必须在执行之前完成它们之间的链接，否则将无法得到正确的内存地址和内存空间。

静态重定位的优点是不需要硬件支持。但是，使用静态重定位方法进行地址变换无法实现虚拟存储器。这是因为，虚拟存储器呈现在用户面前的是一个在物理上只受内存和外存总容量限制的存储系统，这要求存储管理系统只把进程执行时频繁使用和立即需要的指令与数据等存放在内存中，而把那些暂时不需要的部分存放在外存中，待需要时自动调入，以提高内存的利用率和并发执行的作业道数。显然，这是与静态重定位方法矛盾的，静态重定位方法将程序一旦装入内存之后就不能再移动，并且必须在程序执行之前将有关部分全部装入。

静态重定位的另一个缺点是必须占用连续的内存空间，这就难以做到程序和数据的共享。

**2. 动态地址重定位**

动态地址重定位(dynamic address relocation)是在程序执行过程中，在 CPU 访问内存之前，将要访问的程序或数据地址转换成内存地址。动态重定位依靠硬件地址变换机构完成。

地址重定位机构需要一个(或多个)基地址寄存器 BR 和一个(或多个)程序虚拟地址寄存器 VR。指令或数据的内存地址 MA 与虚拟地址的关系为

$$MA=(BR)+(VR)$$

这里，(BR)与(VR)分别表示寄存器 BR 与 VR 中的内容。

动态重定位过程可参看图 5.3。

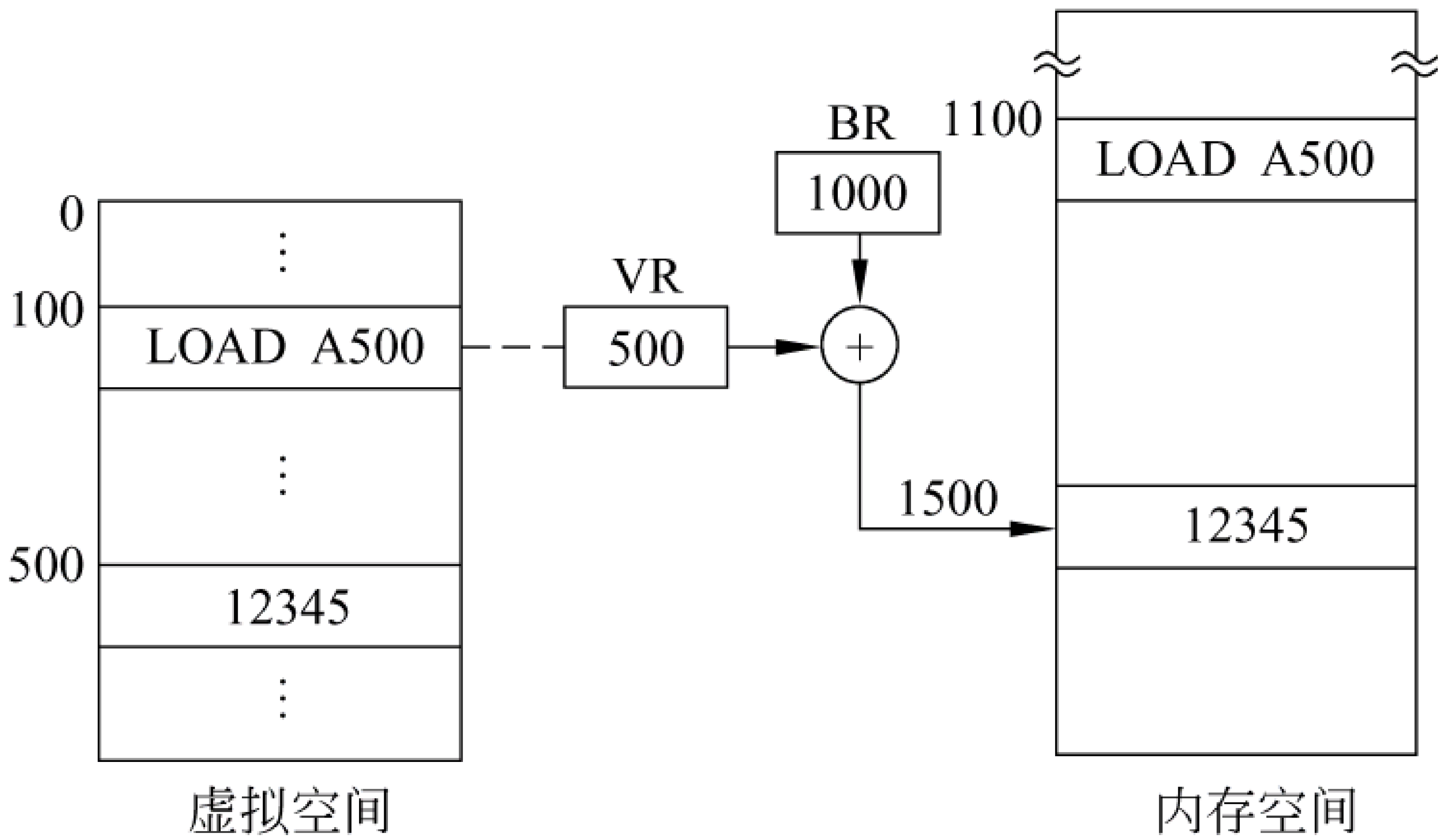



图 5.3　动态地址重定位

其具体过程如下：

(1) 设置基地址寄存器 BR 和虚拟地址寄存器 VR。

(2) 将程序段装入内存，且将其占用的内存区首地址送入 BR 中。例如，在图 5.3 中，(BR)=1000。

(3) 在程序执行过程中，将所要访问的虚拟地址送入 VR 中，例如在图 5.3 中执行 LOAD A 500 语句时，将所要访问的虚拟地址 500 放入 VR 中。

(4) 地址变换机构把 VR 和 BR 的内容相加,得到实际访问的物理地址。

动态重定位的主要优点如下:

(1) 可以对内存进行非连续分配。显然,对于同一进程的各分散程序段,只要把各程序段在内存中的首地址统一存放在不同的 BR 中,就可以由地址变换机构变换得到正确的内存地址。

(2) 动态重定位提供了实现虚拟存储器的基础。因为动态重定位不要求在作业执行前为所有程序分配内存,也就是说,可以部分地、动态地分配内存。从而,可以在动态重定位的基础上,在执行期间采用请求方式为那些不在内存中的程序段分配内存,以达到内存扩充的目的。

(3) 有利于程序段的共享。

### 5.1.3 内外存数据传输的控制

要实现内存扩充,在程序执行过程中,内存和外存之间必须经常地交换数据。也就是说,把那些即将执行的程序和数据段调入内存,而把那些处于等待状态的程序和数据段调出内存。那么,按什么样的方式来控制内存和外存之间的数据流动呢?最基本的控制办法有两种,一种是用户程序自己控制,另一种是操作系统控制。

用户程序自己控制内外存之间的数据交换的例子是覆盖(overlay)。覆盖技术要求用户清楚地了解程序的结构,并指定各程序段调入内存的先后次序。覆盖是一种早期的内存扩充技术。使用覆盖技术,用户负担很大,且程序段的最大长度仍受内存容量限制。因此,覆盖技术不能实现虚拟存储器。

操作系统控制方式又可进一步分为两种,一种是交换(swapping)方式,另一种是请求调入(on demand)方式和预调入(on prefetch)方式。

交换方式由操作系统把那些在内存中处于等待状态的进程换出内存,而把那些等待事件已经发生、处于就绪态的进程换入内存。

请求调入方式是在程序执行时,如果所要访问的程序段或数据段不在内存中,则操作系统自动地从外存将有关的程序段和数据段调入内存的一种操作系统控制方式。而预调入则是由操作系统预测在不远的将来会访问到的那些程序段和数据段部分,并在它们被访问之前选择适当的时机将它们调入内存的一种数据流控制方式。

由于交换方式一般不进行部分交换,即每次交换都交换那些除去常驻内存部分后的整个进程,而且,即使能完成部分交换,也不是按照执行的需要而交换程序段,只是把受资源限制、暂时不能执行的程序段换出内存。因此,虽然交换方式也能完成内存扩充任务,但它仍未实现那种所谓自动覆盖、内存和外存统一管理、进程大小不受内存容量限制的虚拟存储器。只有请求调入方式和预调入方式可以实现进程大小不受内存容量限制的虚拟存储器。有关实现方法将在后面章节中讲述。

### 5.1.4 内存的分配与回收

内存的分配与回收是内存管理的主要功能之一。无论采用哪一种管理和控制方式,能否把外存中的数据和程序调入内存,取决于能否在内存中为它们安排合适的位置。因此,存储管理模块要为每一个并发执行的进程分配内存空间。另外,当进程执行结束之后,存储管

理模块又要及时回收该进程所占用的内存资源，以便给其他进程分配空间。

为了有效合理地利用内存，设计内存的分配和回收方法时，必须考虑和确定以下几种策略和数据结构：

（1）分配结构。登记内存使用情况以及供分配程序使用的表格与链表。例如内存空闲区表、空闲区队列等。

（2）放置策略。确定调入内存的程序和数据在内存中的位置。这是一种选择内存空闲区的策略。

（3）交换策略。在需要将某个程序段和数据段调入内存时，如果内存中没有足够的空闲区，由交换策略来确定把内存中的哪些程序段和数据段调出内存，以便腾出足够的空间。

（4）调入策略。外存中的程序段和数据段什么时间按什么样的控制方式进入内存。调入策略与5.1.3节中所述内外存数据流动控制方式有关。

（5）回收策略。回收策略包括两点，一是回收的时机，二是对所回收的内存空闲区和已存在的内存空闲区的调整。

### 5.1.5 内存信息的共享与保护

内存信息的共享与保护也是内存管理的重要功能之一。在多道程序设计环境下，内存中的许多用户程序或系统程序和数据段可供不同的用户进程共享。这种资源共享将会提高内存的利用率。但是，反过来说，除了被允许共享的部分之外，又要限制各进程只在自己的存储区活动，各进程不能对别的进程的程序和数据段产生干扰和破坏，因此必须对内存中的程序和数据段采取保护措施。

常用的内存信息保护方法有硬件法、软件法和软硬件结合法3种。

上下界保护法是一种常用的硬件保护法。上下界存储保护技术要求为每个进程设置一对上下界寄存器，其中装有被保护程序和数据段的起始地址和终止地址。在程序执行过程中，在对内存进行访问操作时首先进行访址合法性检查，即检查经过重定位后的内存地址是否在上下界寄存器所规定的范围之内。若在规定的范围之内，则访问是合法的；否则是非法的，并产生访址越界中断。上下界保护法的保护原理如图5.4所示。

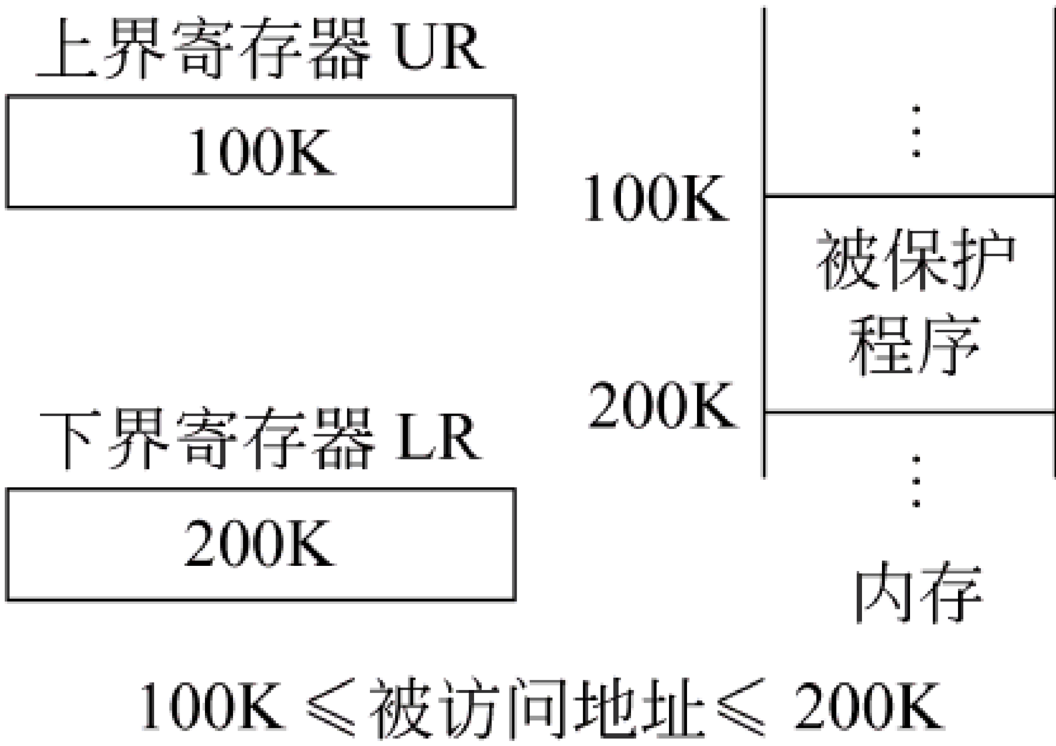



图 5.4　上下界寄存器保护法

另外，保护键法也是一种常用的存储保护法。保护键法为每一个被保护存储块分配一个单独的保护键。在程序状态字中则设置相应的保护键开关字，对不同的进程赋予不同的开关代码与被保护的存储块中的保护键匹配。保护键可设置成对读写同时保护，也可以设置成只对读或写进行单项保护。例如，图5.5中的保护键0就是对2K到4K的存储区进行读写同时保护，而保护键2则只对4K到6K的存储区进行写保护。如果开关字与保护键匹配或存储块未受到保护，则访问该存储块是允许的，否则将产生访问出错中断。

另外一种常用的内存保护方式是界限寄存器与CPU的用户态或核心态工作方式相结合的保护方式。在这种保护方式下，用户态进程只能访问那些在界限寄存器所规定范围内的内存部分，而核心态进程则可以访问整个内存地址空间。UNIX系统就是采用的这种内存保护方式。



图 5.5 保护键保护法

## 5.2 分区存储管理

分区管理是把内存划分成若干个大小不等的区域,除操作系统占用一个区域之外,其余由多道环境下的各并发进程共享。分区管理是满足多道程序设计的一种最简单的存储管理方法。

下面结合分区原理来讨论分区存储管理时的虚存实现、地址变换、内存的分配与释放以及内存信息的共享与保护等问题。

### 5.2.1 分区管理基本原理

分区管理的基本原理是给每一个内存中的进程划分一块适当大小的存储区,以连续存储各进程的程序和数据,使各进程得以并发执行。按分区的时机,分区管理可以分为固定分区和动态分区两种方法。

**1. 固定分区法**

固定分区法就是把内存固定地划分为若干个大小不等的区域。分区划分的原则由一般系统操作员或操作系统决定。例如可划分为长作业分区和短作业分区。分区一旦划分结束,在整个执行过程中每个分区的长度和内存的总分区个数将保持不变。

系统对内存的管理和控制通过数据结构——分区说明表进行,分区说明表说明各分区号、分区大小、起始地址和是否是空闲区(分区状态)。内存的分配释放、存储保护以及地址变换等都通过分区说明表进行。图 5.6 给出了固定分区时分区说明表和对应内存状态的例子。

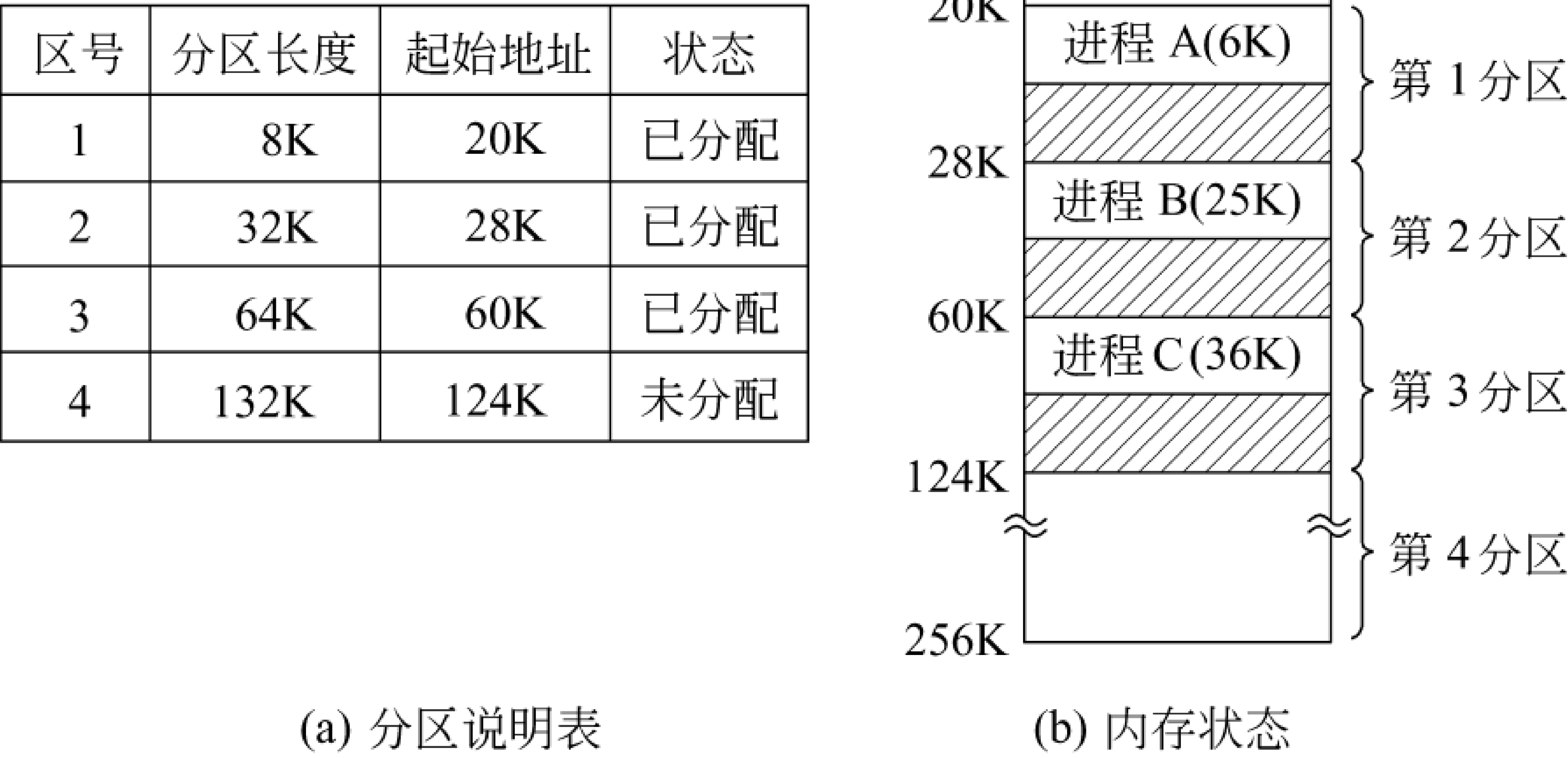
图中,操作系统占用低地址部分的 20K,其余空间被划分为 4 个分区,其中 1、2、3 号分区已分配,4 号分区未分配。

**2. 动态分区法**

与固定分区法相比,动态分区法在作业执行前并不建立分区,分区的建立是在作业的处理过程中进行的,且其大小可随作业或进程对内存的要求而改变。这就改变了固定分区法中那种即使是小作业也要占据大分区的浪费现象,从而提高了内存的利用率。

采用动态分区法,在系统初启时,除了操作系统中常驻内存部分之外,只有一个空闲分区。随后,分配程序将该区依次划分给调度选中的作业或进程。图 5.7 给出了 FIFO 调度方式时的内存初始分配情况。

随着进程的执行,会出现一系列的分配和释放。如在某一时刻,进程 C 执行结束并释



| 区号 | 分区长度 | 起始地址 | 状态 |
|---|---|---|---|
| 1 | 8K | 20K | 已分配 |
| 2 | 32K | 28K | 已分配 |
| 3 | 64K | 60K | 已分配 |
| 4 | 132K | 124K | 未分配 |



(a) 分区说明表　　(b) 内存状态

图 5.6　固定分区法

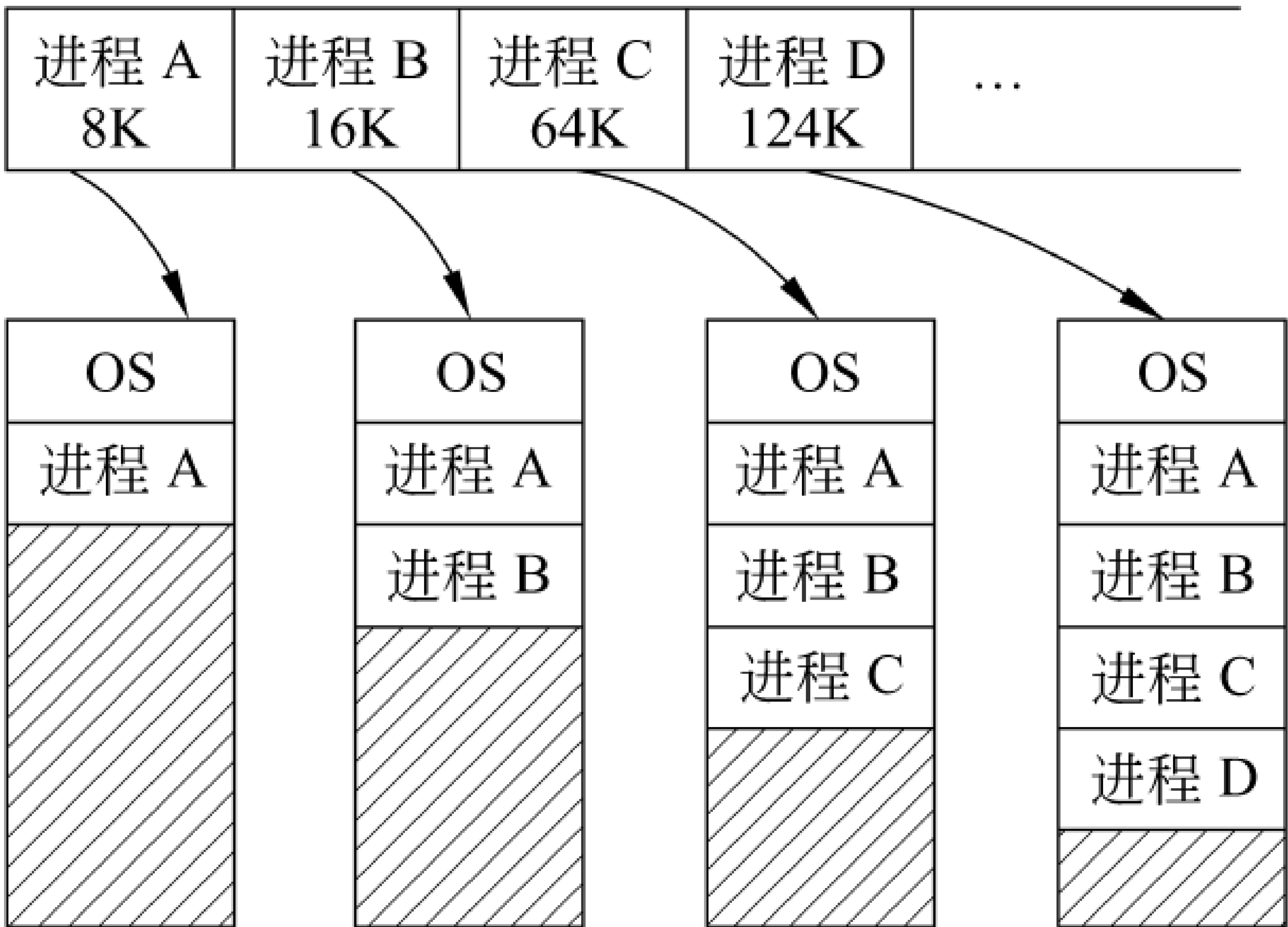




图 5.7　内存初始分配情况

放内存之后，管理程序又要为另两个进程 E(设需内存 50K)和 F(设需内存 16K)分配内存。如果分配的空闲区比所要求的大，则管理程序将该空闲区分成两个部分，其中一部分成为已分配区，而另一部分成为一个新的小空闲区。图 5.8 给出了采用最先适应算法(first fit)分配内存时进程 E 和进程 F 得到内存以及进程 B 和进程 D 释放内存的内存分配变化过程。如图 5.8 所示，在管理程序回收内存时，如果被回收分区有和它邻接的空闲分区存在，则要进行合并。

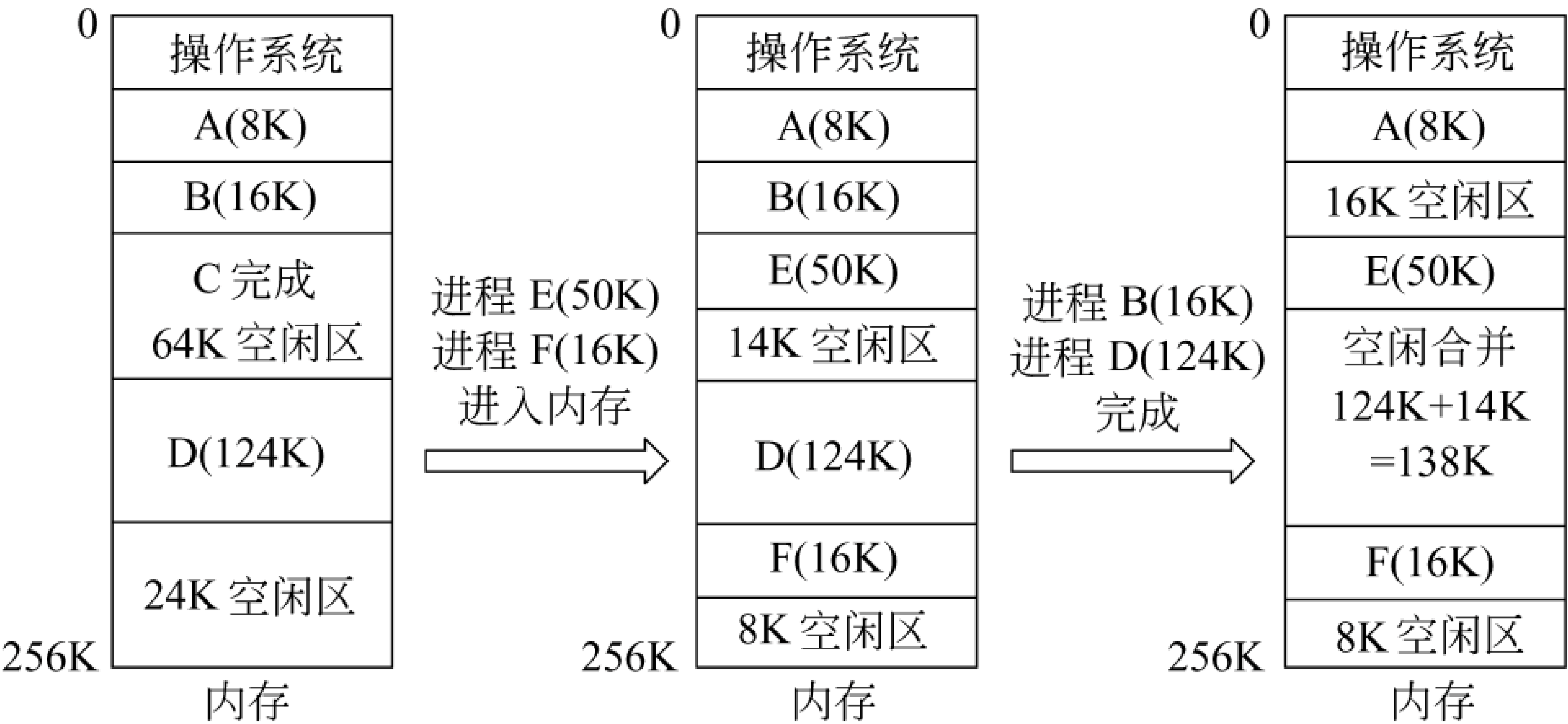




图 5.8　内存分配变化过程

与固定分区法相同，动态分区法也要使用分区说明表等数据结构对内存进行管理。除了分区说明表之外，动态分区法还把内存中的可用分区单独构成可用分区表或可用分区自由链，以描述系统内的内存资源。与此相对应，请求内存资源的作业或进程也构成一个内存资源请求表。图 5.9 给出了可用表、自由链和请求表的例子。

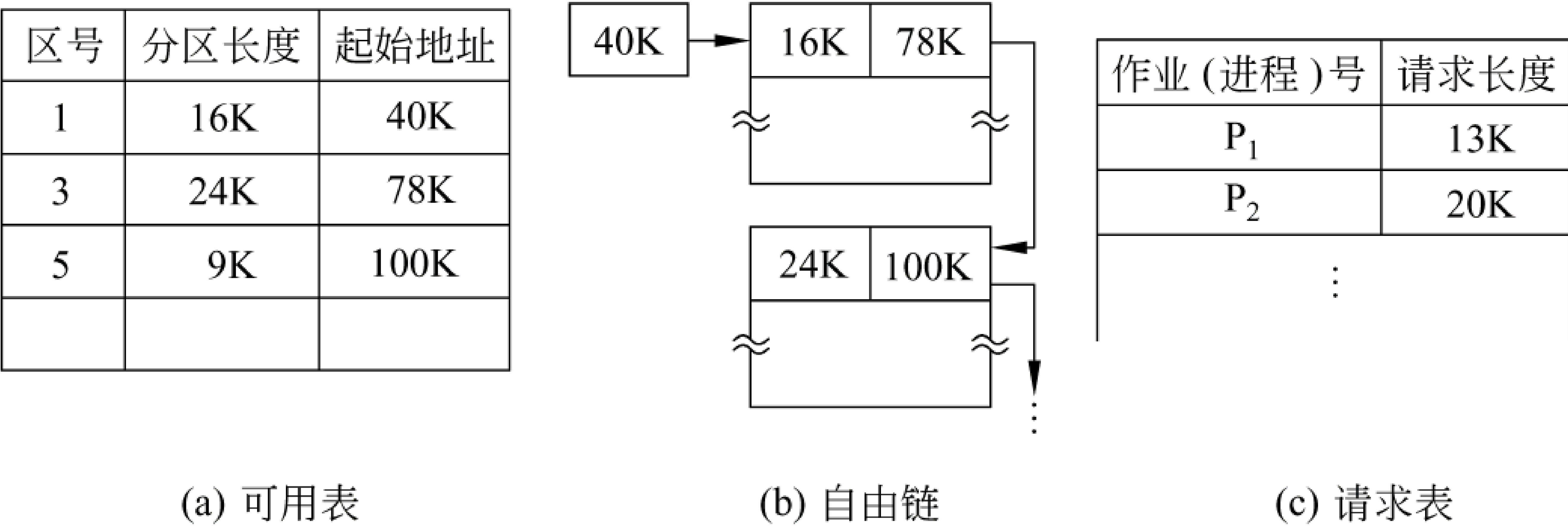



图 5.9　可用表、自由链及请求表

可用表的每个表目记录一个空闲区，主要参数包括区号、分区长度和起始地址。采用表格结构，管理过程比较简单，但表的大小难以确定，要占用一部分内存。

自由链则是利用每个内存空闲区的头几个单元存放本空闲区的大小及下个空闲区的起始地址，从而把所有的空闲区链接起来。然后，系统再设置一个自由链首指针让其指向第一个空闲区，这样，管理程序可通过链首指针查到所有的空闲区。采用自由链法管理空闲区，查找时要比可用表困难，但由于自由链指针是利用空闲区自身的单元，所以不必占用额外的内存区。

请求表的每个表目描述请求内存资源的作业或进程号以及所请求的内存大小。

无论是采用可用表方式还是自由链方式，可用表或自由链中的各个项都要按照一定的规则排列，以利查找和回收。下面讨论分区法的分区分配与回收问题。

## 5.2.2　分区的分配与回收

### 1. 固定分区时的分配与回收

固定分区法时的内存分配与回收较为简单，当用户程序要装入执行时，通过请求表提出内存分配要求和所要求的内存空间大小。存储管理程序根据请求表查询分区说明表，从中找出一个满足要求的空闲分区，并将其分配给申请者。固定分区时的分配算法如图 5.10 所示。

固定分区的回收更加简单。当进程执行完毕，不再需要内存资源时，管理程序将对应的分区状态置为未使用即可。

### 2. 动态分区时的分配与回收

动态分区时的分配与回收主要解决 3 个问题：

(1) 对于请求表中的要求内存长度，从可用表或自由链中寻找出合适的空闲区分配程序。

(2) 分配空闲区之后，更新可用表或自由链。

(3) 进程或作业释放内存资源时，和相邻的空闲区进行链接合并，更新可用表或自由链。

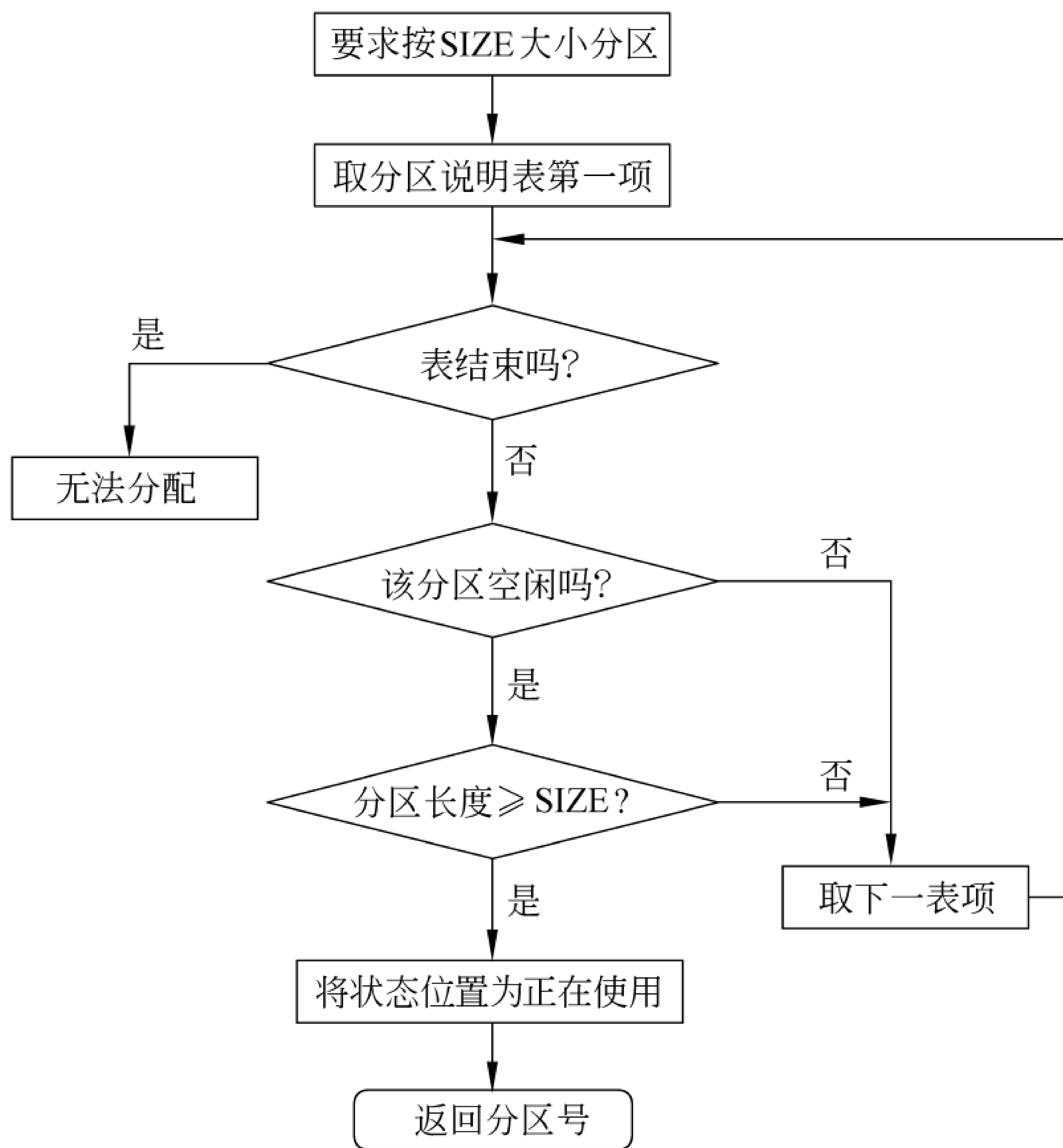



图 5.10　固定分区时的分配算法

从可用表或自由链中寻找空闲区的常用方法有 3 种：最先适应算法(first fit algorithm)、最佳适应算法(best fit algorithm)和最坏适应算法(worst fit algorithm)。这 3 种方法要求可用表或自由链按不同的方式排列。下面分别介绍这 3 种方法。

1）最先适应算法

最先适应算法要求可用表或自由链按起始地址递增的次序排列。该算法的最大特点是一旦找到大于或等于所要求内存长度的分区,则结束探索。然后,该算法从所找到的分区中划出所要求的内存长度分配给用户,并把余下的部分进行合并(如果有相邻空闲区存在)后留在可用表中,但要修改其相应的表项。最先适应算法如图 5.11 所示。

2）最佳适应算法

最佳适应算法要求按从小到大的次序组成空闲区可用表或自由链。当用户作业或进程申请一个空闲区时,存储管理程序从表头开始查找,当找到第一个满足要求的空闲区时,停止查找。如果该空闲区大于请求表中的请求长度,则与最先适应算法时相同,将减去请求长度后的剩余空闲区部分留在可用表中。

3）最坏适应算法

最坏适应算法要求空闲区按其大小递减的顺序组成空闲区可用表或自由链。当用户作业或进程申请一个空闲区时,先检查空闲区可用表或自由链的第一个空闲可用区的大小是否大于或等于所要求的内存长度,若可用表或自由链的第一个项长度小于所要求的,则分配失败,否则从空闲区可用表或自由链中分配相应的存储空间给用户,然后修改空闲区可用表或自由链。

读者可以自行画出最佳适应法和最坏适应法的流程框图。

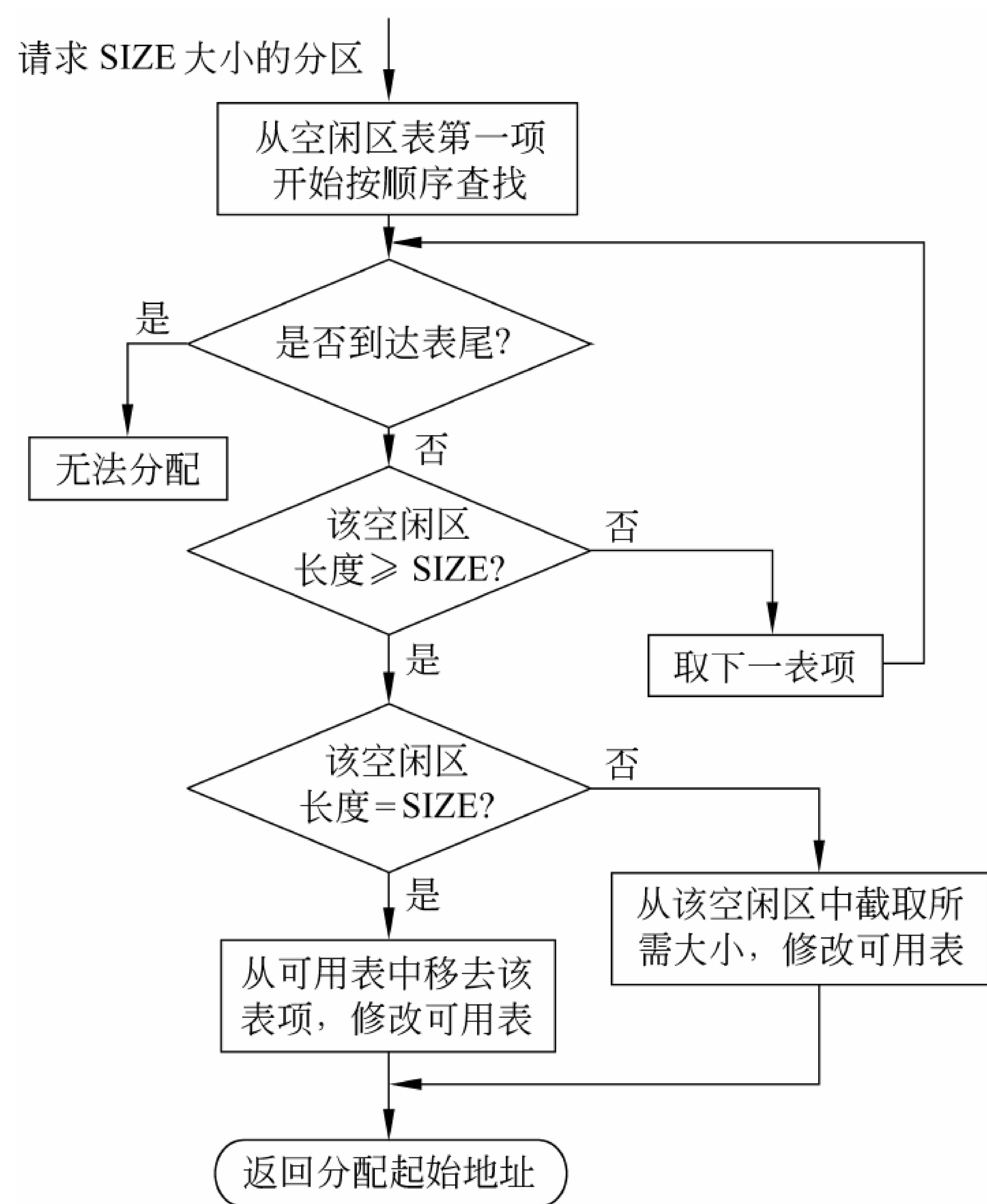



图 5.11　最先适应算法

**3. 动态分区时的回收与拼接**

当用户作业或进程执行结束时，存储管理程序要收回已使用完毕的空闲区(称为释放区)，并将其插入空闲区可用表或自由链。这里，在将回收的空闲区插入可用表或自由链时，和分配空闲区时一样，也要遇到剩余空闲区拼接问题。如果不对空闲区进行拼接，则由于每个作业或进程所要求的内存长度不一样而出现大量分散、较小的空闲区。这就造成大量的内存浪费。解决这个问题的办法之一就是在空闲区回收时或在内存分配时进行空闲区拼接，以把不连续的零散空闲区集中起来。

在将一个新的空闲区插入可用表或自由链时，该空闲区和上下相邻区的关系是下述4种关系之一(如图5.12所示)。

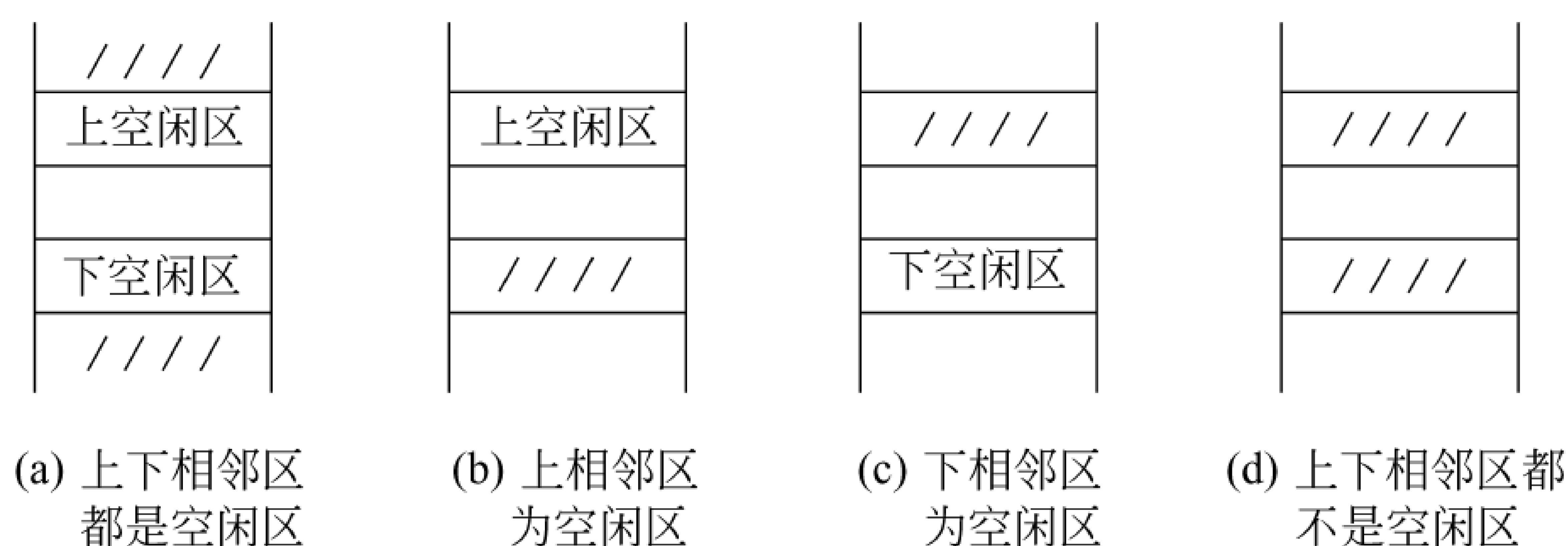



图 5.12　空闲区的合并

(1) 该空闲区的上下两相邻分区都是空闲区。

(2) 该空闲区的上相邻区是空闲区。

（3）该空闲区的下相邻区是空闲区。

（4）该空闲区的上下两相邻区都不是空闲区。

对于上述4种情况，如果释放区与上下两空闲区相邻，则将3个空闲区合并为一个空闲区，新空闲区的起始地址为上空闲区的起始地址，大小为3个空闲区之和。空闲区合并后，取消可用表或自由链中下空闲区的表项或链指针，修改上空闲区的对应项。

如果释放区只与上空闲区相邻，则将释放区与上空闲区合并为一个空闲区，其起始地址为上空闲区的起始地址，大小为上空闲区与释放区之和。合并后，修改上空闲区对应的可用表的表项或自由链指针。

如果释放区与下空闲区相邻，则将释放区与下空闲区合并，并将释放区的起始地址作为合并区的起始地址，合并区的长度为释放区与下空闲区之和。同理，合并后修改可用表或自由链中相应的表项或链指针。

如果释放区不与任何空闲区相邻，则释放区作为一个新的空闲可用区插入可用表或自由链。读者可以自行写出动态分区时的回收算法。

**4. 几种分配算法的比较**

上面讨论了3种常用的内存分配算法及回收算法。由于回收后的空闲区要插入可用表或自由链中，而且可用表或自由链是按照一定顺序排列的，所以，除了搜索查找速度以及所找到的空闲区是否最佳以外，释放空闲区的速度也对系统开销产生影响。下面从查找速度、释放速度及空闲区的利用3个方面对上述3种算法进行比较。

首先，从搜索速度上看，最先适应算法具有最佳性能。尽管最佳适应算法或最坏适应算法看上去能很快地找到一个最适合的或最大的空闲区，但后两种算法都要求首先把大小不同的空闲区按其大小进行排队，这实际上是对所有空闲区进行一次搜索。再者，从回收过程来看，最先适应算法也是最佳的。因为使用最先适应算法回收某一空闲区时，无论被释放区是否与空闲区相邻，都不用改变该区在可用表或自由链中的位置，只需修改其大小或起始地址。而最佳适应算法和最坏适应算法都必须重新调整该区的位置。

最先适应算法的另一个优点就是尽可能地利用了低地址空间，从而保证高地址有较大的空闲区来放置要求内存较多的进程或作业。

反过来，最佳适应法找到的空闲区是最佳的，也就是说，用最佳适应法找到的空闲区或者是正好等于用户请求的大小或者是能满足用户要求的最小空闲区。不过，尽管最佳适应法能选出最适合用户要求的可用空闲区，但这样做在某些情况下并不一定能提高内存的利用率。例如，当用户请求小于最小空闲区不太多时，分配程序会将其分配后的剩余部分作为一个新的小空闲区留在可用表或自由链中。这种小空闲区有可能永远得不到再利用(除非与别的空闲区合并)，而且也会增加内存分配和回收时的查找负担。

最坏适应算法正是基于不留下碎片空闲区这一出发点的，它选择最大的空闲区来满足用户要求，以期分配后的剩余部分仍能进行再分配。

总之，上述3种算法各有特长，针对不同的请求队列，效率和功能是不一样的。

### 5.2.3 有关分区管理其他问题的讨论

**1. 关于虚存的实现**

利用分区式管理，也同样存在每个用户可以自由编程的虚拟空间。但是，分区式管理方

式无法实现那种用户进程所需内存容量只受内存和外存容量之和限制的虚拟存储器。事实上，如果不采用内存扩充技术，每个用户进程所需内存容量是受到分区大小限制的，这一点从上面的讨论中可以看出。

**2. 关于内存扩充**

由于分区式管理时各用户进程或作业所要求的内存容量受到分区大小的限制，如果不采用内存扩充技术，将会极大地限制分区式管理技术的使用。在分区式管理中，可以使用覆盖或交换技术来扩充内存。有关覆盖和交换技术的基本原理，将在5.3节中讨论。

**3. 关于地址变换和内存保护**

静态地址重定位和动态地址重定位技术都可用来完成分区式内存管理的地址变换。显然，动态分区时分区大小不固定，而空闲区的拼接会移动内存中的程序和数据，因此，使用静态地址重定位的方法来完成动态分区时的地址变换是不妥当的。

在进行动态地址重定位时，每个分区需要一对硬件寄存器的支持，即基址寄存器和限长寄存器，分别用来存放作业或进程在内存分区的起始地址和长度。这一对硬件寄存器除了完成动态地址重定位的功能之外，还具有保护内存中数据和程序的功能。这由硬件检查CPU执行指令所要访问的虚拟地址完成。即设CPU指令所要访问的虚拟地址为$D$，若$D>VR$（VR是限长寄存器中的限长值），则说明地址越界，所要访问的内存地址超出了该作业或进程所占用的内存空间。这时将产生保护中断，系统转去进行出错处理。若$D\leqslant VR$，则该地址是合法的，由硬件完成对该虚拟地址的动态重定位。

保护键法也可用来对内存各分区提供保护。

**4. 分区存储管理的主要优缺点**

分区存储管理的主要优点如下：

（1）实现了多个作业或进程对内存的共享，有助于多道程序设计，从而提高了系统的资源利用率。

（2）该方法要求的硬件支持少，管理算法简单，因而实现容易。

其主要缺点如下：

（1）内存利用率仍然不高。和单一连续分配算法一样，内存中可能含有从未用过的信息。而且，还存在着严重的碎小空闲区（碎片）不能利用的问题，这更进一步影响了内存的利用率。

（2）作业或进程的大小受分区大小控制，除非配合采用覆盖和交换技术。

（3）无法实现各分区间的信息共享。

## 5.3 覆盖与交换技术

覆盖与交换技术是在多道环境下用来扩充内存的两种方法。覆盖技术主要用在早期的操作系统中，而交换技术则在现代操作系统中仍具有较强的生命力。下面主要介绍覆盖与交换的基本思想。

### 5.3.1 覆盖技术

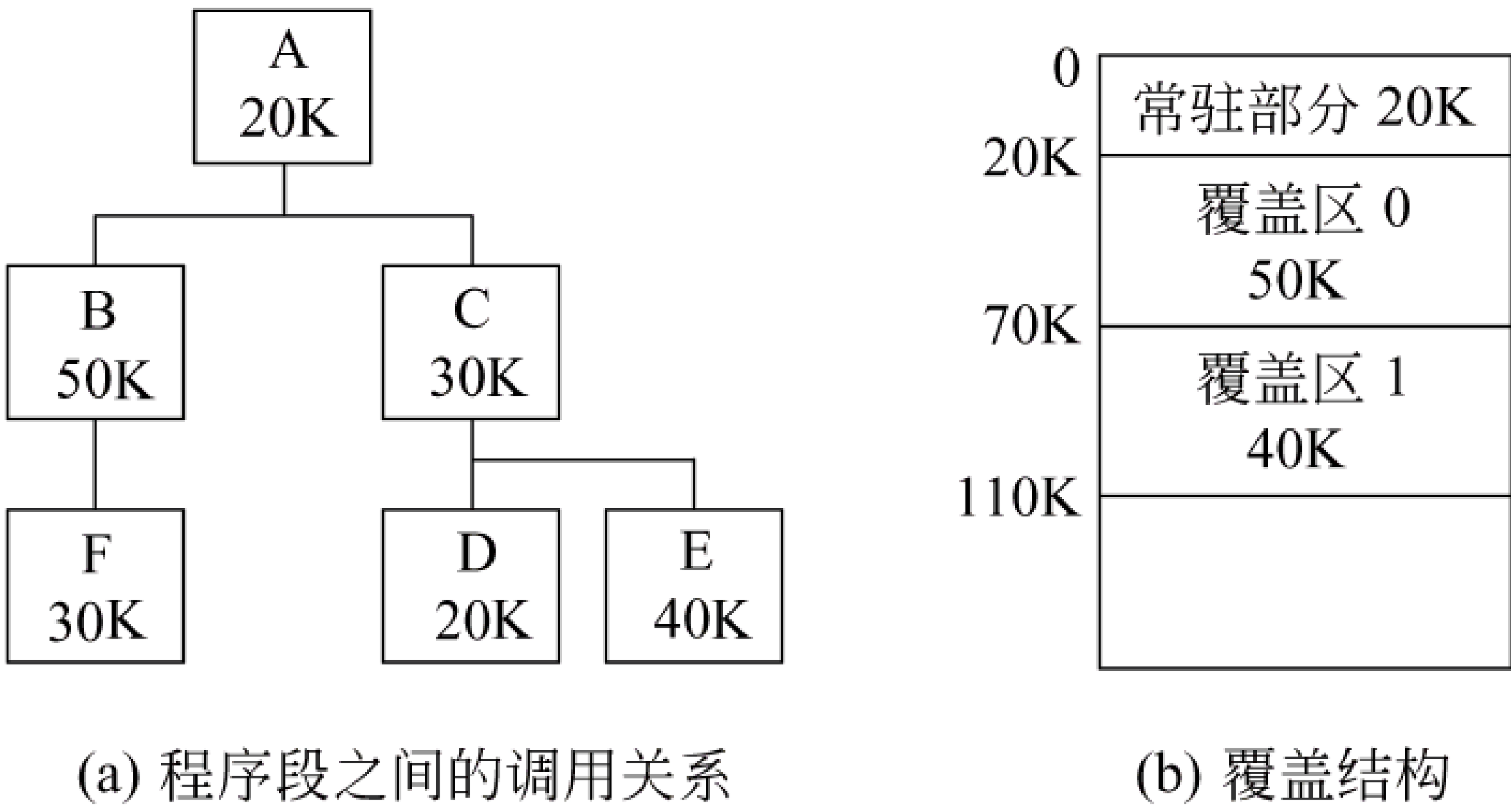
覆盖技术是基于这样一种思想提出来的，即一个程序并不需要一开始就把它的全部指

令和数据都装入内存后再执行。在单 CPU 系统中，每一时刻事实上只能执行一条指令。因此，不妨把程序划分为若干个功能上相对独立的程序段，按照程序的逻辑结构让那些不会同时执行的程序段共享同一块内存区。通常，这些程序段都被保存在外存中，当有关程序段的先头程序段执行结束后，再把后续程序段调入内存覆盖前面的程序段。这使得用户看来，好像内存扩大了，从而达到了内存扩充的目的。

但是，覆盖技术要求程序员提供一个清楚的覆盖结构，即程序员必须完成把一个程序划分成不同的程序段，并规定好它们的执行和覆盖顺序的工作。操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖。一般来说，一个程序究竟可以划分为多少段，以及让其中的哪些程序共享哪一内存区只有程序员清楚。这要求程序员既要清楚地了解程序所属进程的虚拟空间及各程序段所在虚拟空间的位置，又要求程序员懂得系统和内存的内部结构与地址划分，因此，程序员负担较重。所以，覆盖技术大多限于对操作系统的虚空间和内部结构很熟悉的程序员使用。

例如，设某进程的程序正文段由 A、B、C、D、E 和 F 共 6 个程序段组成。它们之间的调用关系如图 5.13(a)所示，程序段 A 调用程序段 B 和 C，程序段 B 又调用程序段 F，程序段 C 调用程序段 D 和 E。

由图 5.13(a)可以看出，程序段 B 不会调用 C，程序段 C 也不会调用 B。因此，程序段 B 和 C 无须同时驻留在内存，它们可以共享同一内存区。同理，程序段 D、E、F 也可共享同一内存区。其覆盖结构如图 5.13(b)所示。



(a) 程序段之间的调用关系　　(b) 覆盖结构

图 5.13　覆盖示例

在图 5.13(b)中，整个程序正文段被分为两个部分。一个是常驻内存部分，该部分与所有的被调用程序段有关，因而不能被覆盖。这一部分称为根程序。图 5.13(b)中，程序段 A 是根程序。另一部分是覆盖部分，图中被分为两个覆盖区。其中，一个覆盖区由程序段 B 和 C 共享。其大小为 B、C 中所要求容量大者。另一个覆盖区为程序段 F、D、E 共享。两个覆盖区的大小分别为 50K 与 40K。这样，虽然该进程正文段所要求的内存空间是 20K＋50K＋30K＋30K＋20K＋40K＝190K，但由于采用了覆盖技术，只需 110K 的内存空间即可开始执行。

## 5.3.2　交换技术

在多道程序环境或分时系统中，同时执行好几个作业或进程。但是，这些同时存在于内存中的作业或进程，有的处于执行状态或就绪状态，而有的则处于等待状态。

一般来说，等待时间比较长。例如从外存软磁盘读一块数据到内存有时要花0.1s到1s左右的时间。如果让这些等待中的进程继续驻留内存，将会造成存储空间的浪费。因此，应该把处于等待状态的进程换出内存。

实现上述目标的方法很多，其中比较常用的方法之一就是交换。广义地说，交换是指先将内存某部分的程序或数据写入外存交换区，再从外存交换区中调入指定的程序或数据到内存中来，并让其执行的一种内存扩充技术。与覆盖技术相比，交换不要求程序员给出程序段之间的覆盖结构。而且，交换主要是在进程或作业之间进行，而覆盖则主要在同一个作业或进程内进行。另外，只能对那些与覆盖程序段无关的程序段进行覆盖。

交换进程由换出和换入两个过程组成。其中换出(swap out)过程把内存中的数据和程序换到外存交换区，而换入(swap in)过程把外存交换区中的数据和程序换到内存分区中。

换出过程和换入过程都要完成与外存设备管理进程通信的任务。由交换进程发送给设备进程的消息 $m$ 中应包含分区的分区号 $i$、该分区的基址 $base_i$、长度 $size_i$ 和方向及外存交换区中分区起始地址。交换进程和设备管理进程通过设备缓冲队列进行通信。换出过程SWAPOUT可描述如下：

```
SWAPOUT(i):
begin local m
    m.base←basei;
    m.ceiling←basei+sizei;
    m.direction←"out";
    m.destination←base of free area on swap area;
    backupstorebasei←m.destination;
    send((m,i),device queue);
end
```

在SWAPOUT($i$)中，除了前5行描述所需要的控制信息之外，$backupstorbase_i$ 是用来记录被换出数据和程序的起始地址以便换入时使用的。而send指令则驱动设备做相应的数据读写操作。

与SWAPOUT过程相同，可以写出SWAPIN过程：

```
SWAPIN(i):
begin local m
        m.base←basei;
        m.ceiling←basei+sizei;
        m.direction←"in";
        m.source←backupstorebasei;
        send((m,i),device queue);
end
```

交换技术大多用在小型机或微机系统中，这样的系统大部分采用固定的或动态分区方式管理内存。

# 5.4 页式管理

## 5.4.1 页式管理的基本原理

上面几节中，已经讨论了分区式存储管理方法及支持分区式管理的覆盖与交换技术。事实上，分区式管理方式尽管实现方式较为简单，但存在着严重的碎片问题使得内存的利用率不高。再者，分区式管理时，由于各作业或进程对应于不同的分区以及在分区内各作业或进程连续存放，进程的大小仍受分区大小或内存可用空间的限制。而且，分区式管理也不利于程序段和数据的共享。页式管理正是为了减少碎片以及为了只在内存存放那些反复执行或即将执行的程序段与数据部分，而把那些不经常执行的程序段和数据存放于外存，待执行时调入，以提高内存利用率而提出来的。

页式管理的基本原理如下：

首先，各进程的虚拟空间被划分成若干个长度相等的页(page)。页长的划分和内存、外存之间的数据传输速度以及内存大小等有关。一般每个页长大约为 1K～4K，经过页划分之后，进程的虚地址变为页号 P 与页内地址 W 所组成。例如，一个页长为 1K，拥有 1024 页的虚拟空间地址结构如图 5.14 所示。

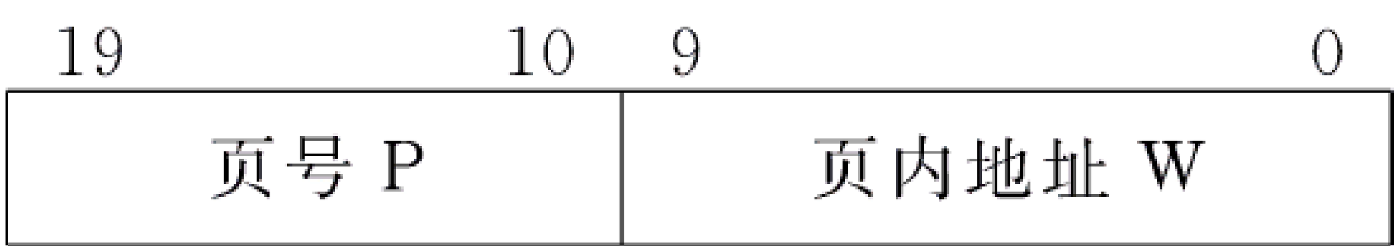



图 5.14 页的划分

除了把进程的虚拟空间划分为大小相等的页之外，页式管理还把内存空间也按页的大小划分为片或页面(page frame)。这些页面为系统中的任一进程所共享(除去操作系统区外)。从而，与分区管理不一样，分页管理时，用户进程在内存空间内除了在每个页面内地址连续之外，每个页面之间不再连续。页式管理有以下两个优点。第一是实现了内存中碎片的减少，因为任一碎片都会小于一个页面。第二是实现了由连续存储到非连续存储这个飞跃，为在内存中局部地、动态地存储那些反复执行或即将执行的程序和数据段打下了基础。

那么，怎样由页式虚拟地址变换为内存页面物理地址呢？页式管理把页式虚地址与内存页面物理地址建立一一对应的页表，并用相应的硬件地址变换机构来解决离散地址变换问题。页表方式实质上是动态重定位技术的一种延伸，这一点在后面的介绍中可以看到。

再者，页式管理采用请求调页或预调页技术实现了内外存储器的统一管理。即内存内只存放那些经常被执行或即将被执行的页，而那些不常被执行以及在近期内不可能被执行的页，则存放于外存中待需要时再调入。请求调页或预调页技术是基于工作区的局部性原理的，有关局部性原理也将在本节的后面部分介绍。

由于使用了请求调页或预调页技术，页式管理时内存页面的分配与回收已和页面淘汰技术及缺页处理技术结合起来。不过，页面的换入换出仍是必要的，这只要把 5.3.2 节所述的交换进程稍加修改就能用于页面的交换。

分页管理的重点在于页划分之后的地址变换以及页面的调入调出技术。

## 5.4.2 静态页式管理

静态页面管理方法是在作业或进程开始执行之前，把该作业或进程的程序段和数据全

部装入内存的各个页面中，并通过页表(page mapping table)和硬件地址变换机构实现虚拟地址到内存物理地址的地址映射。

**1. 内存页面分配与回收**

静态分页管理的第一步是为要求内存的作业或进程分配足够的页面。系统依靠存储页面表、请求表以及页表来完成内存的分配工作。首先来介绍这3个表。

1) 页表

最简单的页表由页号与页面号组成，如图5.15所示。

页表在内存中占有一块固定的存储区。页表的大小由进程或作业的长度决定。例如，对于一个每页长1K，大小为20K的进程来说，如果一个内存单元存放一个页表项，则只要分配给该页表20个存储单元即可。显然，页式管理时每个进程至少拥有一个页表。

2) 请求表

请求表用来确定作业或进程的虚拟空间的各页在内存中的实际对应位置。为了完成这个任务，系统必须知道每个作业或进程的页表起始地址和长度，以进行内存分配和地址变换。另外，请求表中还应包括每个作业或进程所要求的页面数。

整个系统需要一张请求表，如图5.16所示。

| 页号 | 页面号 |
|---|---|
| | |
| | |
| | |

图5.15　基本页表示例

| 进程号 | 请求页面数 | 页表始址 | 页表长度 | 状态 |
|---|---|---|---|---|
| 1 | 20 | 1024 | 20 | 已分配 |
| 2 | 34 | 1044 | 34 | 已分配 |
| 3 | 18 | 1078 | 18 | 已分配 |
| 4 | 21 | ⋮ | ⋮ | 未分配 |
| ⋮ | ⋮ | | | ⋮ |

图5.16　请求表示例

3) 存储页面表

存储页面表也是整个系统一张，存储页面表指出内存各页面是否已被分配出去，以及未分配页面的总数。存储页面表也有两种构成方法，一种是在内存中划分一块固定区域，每个单元的每个比特代表一个页面。如果该页面已被分配，则对应比特位置1，否则置0，这种方法称为位示图法，如图5.17所示。

| 19 | 18 | 17 | 16 | 15 | … | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | … | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | … | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | … | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | | | |

图5.17　位示图

位示图要占据一部分内存容量，例如，一个划分为1024个页面的内存，如果内存单元长20bit，则位示图要占据1024/20=52个内存单元。

存储页面表的另一种构成办法是采用空闲页面链的方法。在空闲页面链中，队首页面的第一个单元和第二个单元分别放入空闲页面总数与指向下一个空闲页面的指针。其他页面的第一个单元中则分别放入指向下一个页面的指针。空闲页面链的方法由于使用了空闲页面本身的单元存放指针，因此不占据额外的内存空间。

**2. 分配算法**

利用上述 3 个表格和数据结构，给出一个页面分配算法。

首先，请求表给出进程或作业要求的页面数。然后，由存储页面表检查是否有足够的空闲页面，如果没有，则本次无法分配；如果有，则首先分配设置页表，并填写请求表中的相应表项后，按一定的查找算法搜索出所要求的空闲页面，并将对应的页面号填入页表中。图 5.18 给出了上述页面分配算法的流程图。

静态页式管理的页面回收方法较为简单，当进程执行完毕时，拆除对应的页表，并把页表中的各页面插入存储页面表即可。

**3. 地址变换**

静态页式管理的另一个关键问题是地址变换，即怎样由页号和页内相对地址变换到内存物理地址的问题。另外，由于静态重定位可以使 CPU 直接访问物理地址，而分区式管理中的动态重定位也只需把基址寄存器中的分区起始地址与待访问指令的虚地址相加即可得到所要访问的物理地址。这两种重定位法都不需要访问内存就可得到待执行指令所要访问的物理地址。那么，页式管理时，地址变换的速度怎样呢？这也是设计地址变换机构时必须考虑的问题之一。

由地址分配方式知道，在一个作业或进程的页表中，连续的页号对应于不连续的页面号。例如，设一个 3 页长的进程具有页号 0、1、2，但其对应的页面号则为 2、3、8，如图 5.19 所示。设每个页面长度为 1K，指令 LOAD 1,2500 的虚地址为 100，怎样通过图 5.19 所示的页表来找到该指令所对应的物理地址呢？下面使用该例子说明地址变换过程。

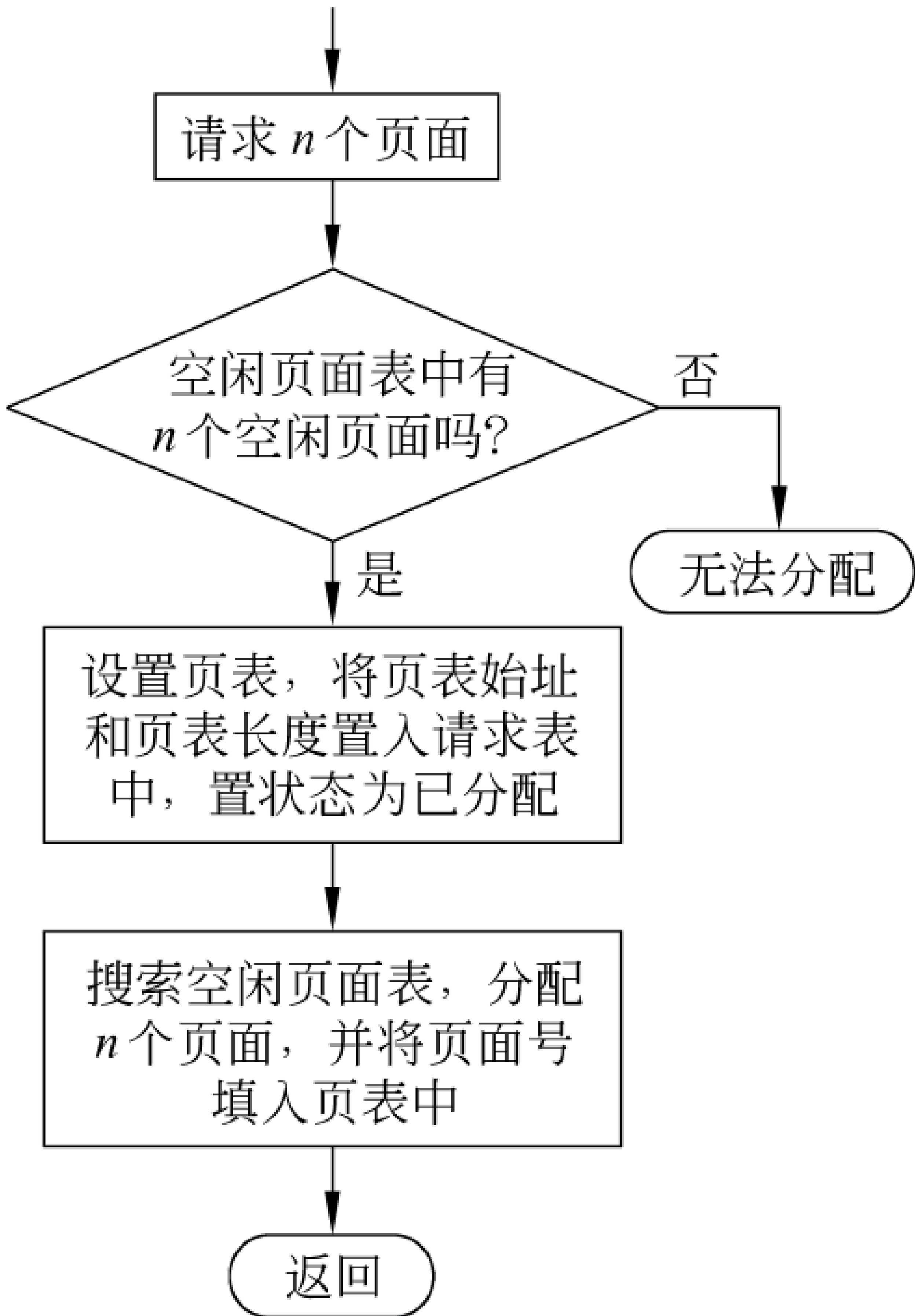



图 5.18　页面分配算法流图

| 页号 | 页面号 |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | 8 |

图 5.19　页号与页面号

首先，需要有一个装置页表始址和页表长度用的控制寄存器。系统把所调度执行的进程页表始址和长度从请求表中取出置入控制寄存器中。

然后，由控制寄存器的页表始址可以找到页表所在位置，并由虚地址 100 可知，指令 LOAD 1,2500 在第 0 页的第 100 单元之中。由于第 0 页与第 2 个页面相对应，因此，该指令在内存中的地址为 2048＋100＝2148。

当 CPU 执行到第 2148 单元的指令时，CPU 要从有效地址 2500 中取数据放入 1 号寄存器中。为了找出 2500 对应的实际物理地址，地址变换机构首先将 2500 转换为页号与页内相对地址组成的地址形式，即 p＝2，w＝452。

由页表可知，第 2 页所对应的页面号等于 8。最后，将页面号 8 与页内相对地址 w＝452 相连，得到待访问的物理内存地址 8644。其变换过程如图 5.20 所示。

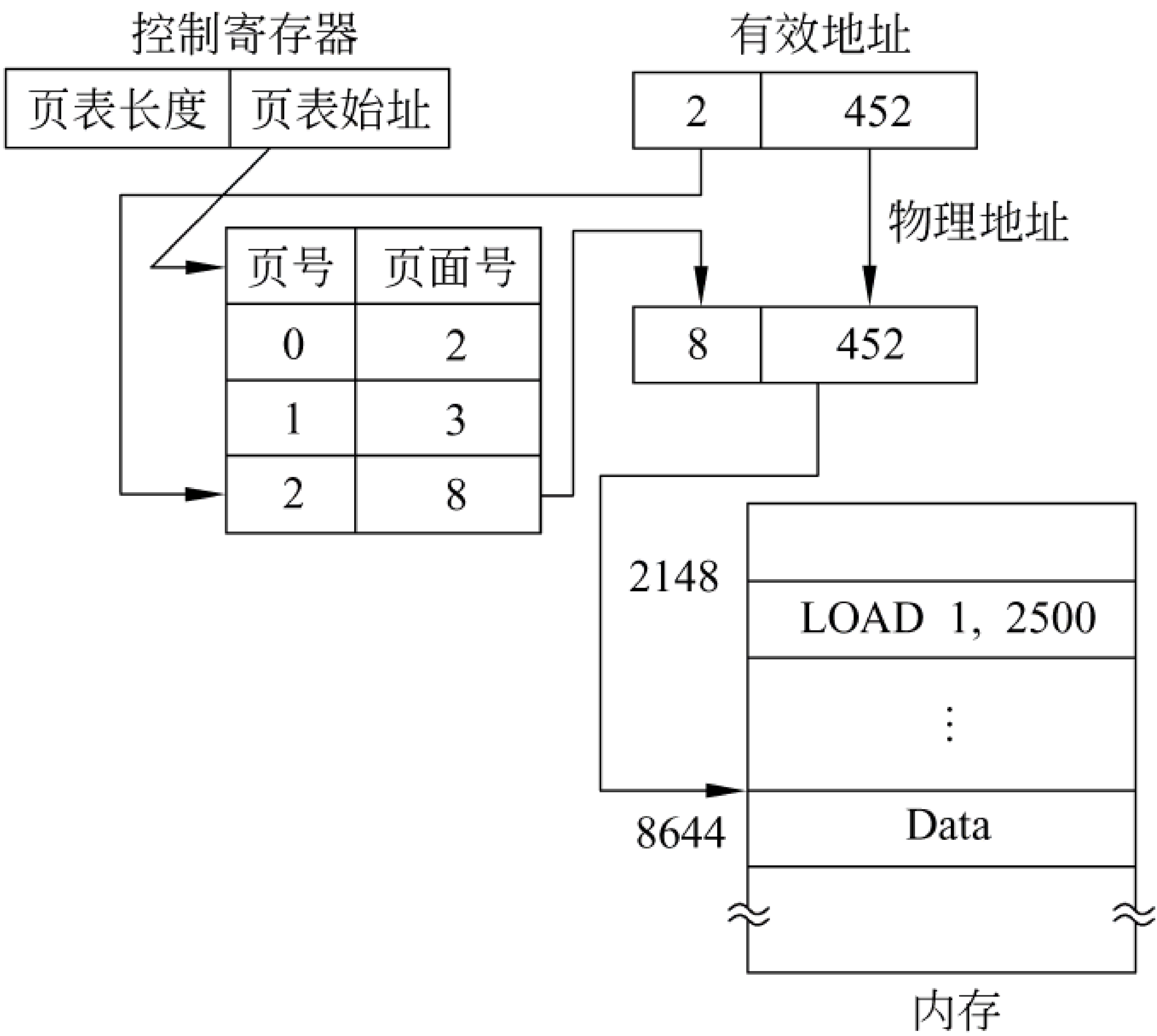



图 5.20　地址变换

上述地址变换过程全部由硬件地址变换机构自动完成。

另外，由于页表是驻留在内存的某个固定区域中，而取数据或指令又必须经过页表变换才能得到实际物理地址。因此，取一个数据或指令至少要访问内存两次以上。一次访问页表以确定所取数据或指令的物理地址，另一次是根据地址取数据或指令。这比通常执行指令的速度慢了一倍。有什么办法可以提高查找速度吗？一个最直观的办法就是把页表放在寄存器中而不是内存中，但由于寄存器价格太贵，这样做是不可取的。另一种办法是在地址变换机构中加入一个高速联想存储器，构成一张快表。在快表中，存入那些当前执行进程中最常用的页号与所对应的页面号，从而提高查找速度。

静态页式管理解决了分区管理时的碎片问题。但是，由于静态页式管理要求进程或作业在执行前全部装入内存，如果可用页面数小于用户要求时，该作业或进程只好等待。而且，作业或进程的大小仍受内存可用页面数的限制。这些问题将在动态（请求）页式管理中解决。

### 5.4.3　动态页式管理

动态页式管理是在静态页式管理的基础上发展起来的。它分为请求页式管理和预调入

页式管理。

请求页式管理和预调入页式管理在作业或进程开始执行之前,都不把作业或进程的程序段和数据段一次性地全部装入内存,而只装入被认为是经常反复执行和调用的工作区部分。其他部分则在执行过程中动态装入。请求页式管理与预调入页式管理的主要区别在它们的调入方式上。请求页式管理的调入方式是,当需要执行某条指令而又发现它不在内存时,或当执行某条指令需要访问其他的数据或指令,而这些指令和数据不在内存中时,即发生缺页中断,系统将外存中相应的页面调入内存。

预调入页式管理的调入方式是,系统对那些在外存中的页进行调入顺序计算,估计出这些页中指令和数据的执行和被访问的顺序,并按此顺序将它们顺次调入和调出内存。除了在调入方式上请求页式管理和预调入页式管理有些区别之外,在其他方面这两种方式基本相同。因此,下面主要介绍请求页式管理。

请求页式管理的地址变换过程与静态页式管理时的相同,也是通过页表查出相应的页面号之后,由页面号与页内相对地址相加而得到实际物理地址。

但是,由于请求页式管理只让进程或作业的部分程序和数据驻留在内存中,因此,在执行过程中,不可避免地会出现某些虚页不在内存中的问题。怎样发现这些不在内存中的虚页以及怎样处理这种情况,是请求页式管理必须解决的两个基本问题。

第一个问题可以用扩充页表的方法解决。即与每个虚页号相对应,除了页面号之外,再增设该页是否在内存的中断位以及该页在外存中的副本起始地址。扩充后的页表如图 5.21 所示。

关于虚页不在内存时的处理涉及两个问题。第一,采用何种方式把所缺的页调入内存。第二,如果内存中没有空闲页面时,把调进来的页放在什么地方。也就是说,采用什么样的策略来淘汰已占据内存的页。还有,如果在内存中的某一页被淘汰,且该页曾因程序的执行而被修改,则显然该页是应该重新写到外存上加以保存的。而那些未被访问修改的页,因为外存已保留有相同的副本,写回外存是没有必要的。因此,在页表中还应增加一项以记录该页是否曾被改变。增加改变位后的页表如图 5.22 所示。

| 页号 | 页面号 | 中断位 | 外存始址 |
| --- | --- | --- | --- |
| 0 | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |

图 5.21　加入中断处理后的页表

| 页号 | 页面号 | 中断位 | 外存始址 | 改变位 |
| --- | --- | --- | --- | --- |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

图 5.22　加入改变位后的页表

有关缺页的调入和存放,在内存中没有空闲页面时,实际上是一个内存页面置换算法的问题。选择什么样的置换算法,将直接影响到内存利用率和系统效率。事实上,如果置换算法选择不当,有可能产生刚被调出内存的页又要马上被调回内存,调回内存不久又马上被调出内存,如此反复的局面。这使得整个系统的页面调度非常频繁,以致大部分时间都花费在内存和外存之间的来回调入调出上,这种现象被称为抖动(thrashing)现象。

有关抖动现象的讨论,将在 5.4.4 节中介绍。

动态页式管理的流程图如图5.23所示。

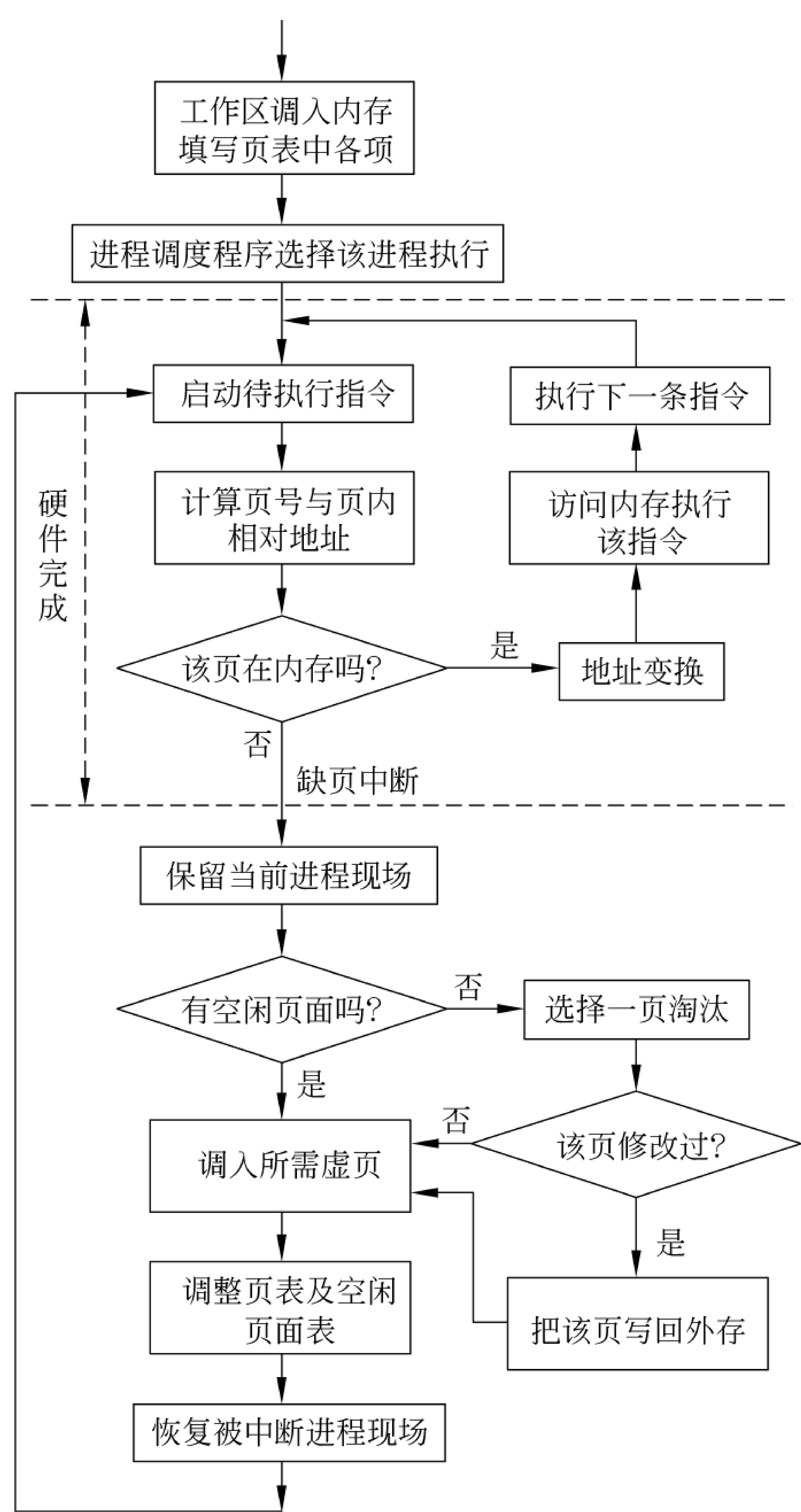



图5.23 动态页式管理流程图

在图5.23中,有关地址变换部分是由硬件自动完成的。当硬件变换机构发现所要求的页不在内存时,产生缺页中断信号,由中断处理程序做出相应的处理。中断处理程序是由软件实现的。除了在没有空闲页面时要按照置换算法选择出被淘汰页面之外,还要从外存读入所需要的虚页。这个过程要启动相应的外存并涉及文件系统。因此,请求页式管理是一个十分复杂的处理过程,内存利用率的提高是以牺牲系统开销的代价换来的。

下面介绍几种常用的置换算法。

### 5.4.4 请求页式管理中的置换算法

置换算法在内存中没有空闲页面时被调用,它的目的是选出一个被淘汰的页面。如果内存中有足够的空闲页面存放所调入的页,则不必使用置换算法。把内存和外存统一管理的真正目的是把那些被访问概率非常高的页存放在内存中。因此,置换算法应该置换那些

被访问概率最低的页，将它们移出内存。比较常用的置换算法有以下几种：

**1. 随机淘汰算法**

在系统设计人员认为无法确定哪些页被访问的概率较低时，随机地选择某个用户的页面并将其换出将是一种明智的做法。

**2. 轮转法(round robin)和先进先出(FIFO)算法**

轮转法循环换出内存可用区内一个可以被换出的页，无论该页是刚被换进或已换进内存很长时间。

FIFO算法总是选择在内存驻留时间最长的一页将其淘汰。FIFO算法认为先调入内存的页不再被访问的可能性要比其他页大，因而选择最先调入内存的页换出。实现FIFO算法需要把各个已分配页面按分配时间顺序链接起来，组成FIFO队列，并设置一个置换指针指向FIFO队列的队首页面。这样，当要进行置换时，只需把置换指针所指的FIFO队列前头的页顺次换出，而把换入的页链接在FIFO队尾即可。

由实验和测试发现FIFO算法和轮转法的内存利用率不高。这是因为，这两种算法都是基于CPU按线性顺序访问地址空间的这个假设。事实上，许多时候，CPU不是按线性顺序访问地址空间的，例如在执行循环语句时。因此，那些在内存中停留时间最长的页往往也是经常被访问的页。尽管这些页变“老”了。但它们被访问的概率仍然很高。

先进先出算法的另一个缺点是它有一种陷阱现象。一般来说，对于任一作业或进程，如果给它分配的内存页面数越接近于它所要求的页面数，则发生缺页的次数会越少。在极限情况下，这个推论是成立的。因为如果给一个进程分配了它所要求的全部页面，则不会发生缺页现象。但是，使用FIFO算法时，在未给进程或作业分配足它所要求的页面数时，有时会出现分配的页面数增多，缺页次数反而增加的奇怪现象。这种现象称为Belady现象，如图5.24所示。

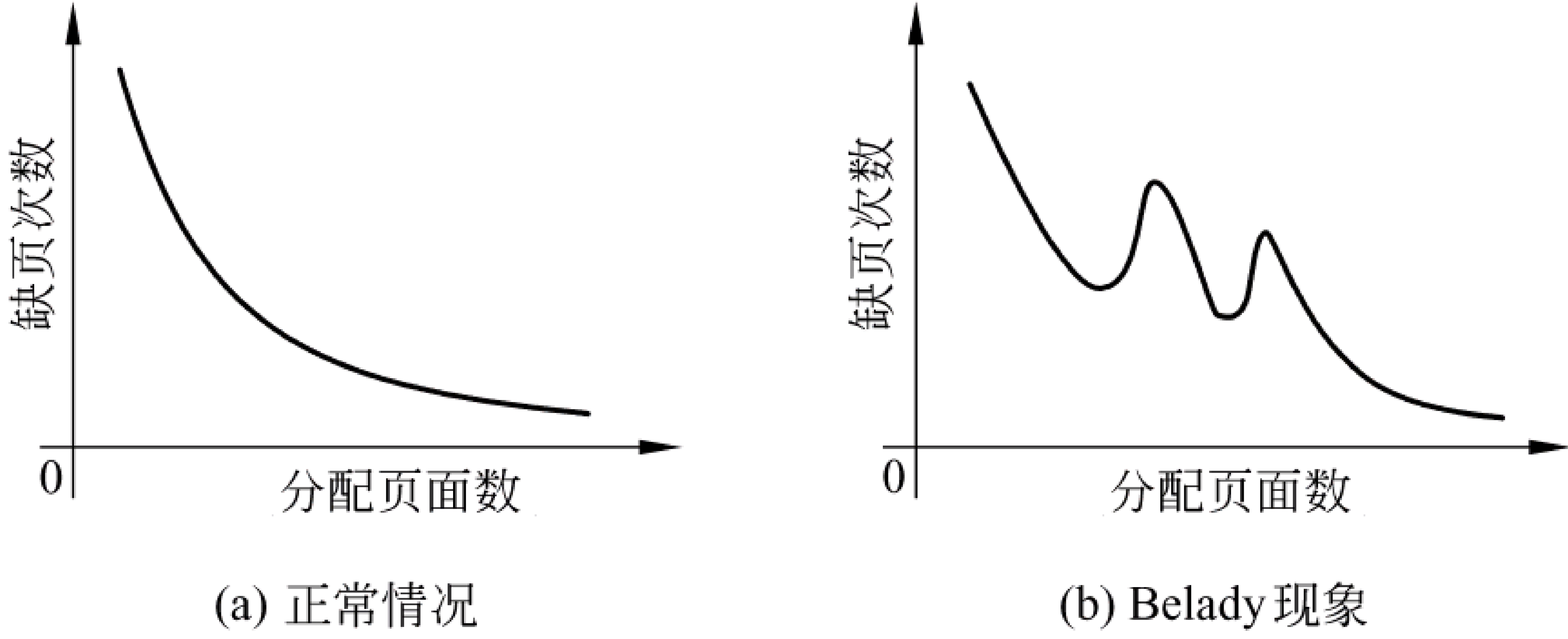



图5.24　FIFO算法的Belady现象

下面的例子可以用来说明FIFO算法的正常换页情况和Belady现象。

设进程P共有8页，且已在内存中分配有3个页面，程序访问内存的顺序(访问串)为7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1。这里，这些自然数代表进程P所建的程序和数据的页号。内存中有关进程P所建的程序和数据的各页面变化情况如图5.25所示。

由图5.25可以看出，进程在一个执行过程中，实际上发生了12次缺页。如果设缺页率为缺页次数与访问串的访问次数之比，则该例中的缺页率为12/17≈70.5%。

如果给进程P分配4个页面，则在其执行过程中内存页面的变化情况如图5.26所示。进程P在拥有4个内存页面时，共发生9次缺页，其缺页率为9/17≈52.9%。

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 |

图 5.25　Belady 现象示例(1)

以上是使用 FIFO 算法正常换页时的例子。下面我们来看看另外一种访问串时的情况。设进程 P 可分为 5 页，访问串为 1，2，3，4，1，2，5，1，2，3，4，5。当进程 P 分得 3 个页面时，执行过程中内存页面变化如图 5.27 所示。

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |

图 5.26　Belady 现象示例(2)

| 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
| | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 |
| | | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 4 | 4 |

图 5.27　Belady 现象示例(3)

由图 5.27 可知，进程 P 在执行过程中共缺页 9 次，其缺页率为 9/12＝75％。

但是，如果为进程 P 分配 4 个内存页面，是否缺页率会变小呢？进程 P 分得 4 个页面时，执行过程中内存页面的变化情况如图 5.28 所示。

| 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 |
| | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 5 |
| | | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |
| | | | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 |

图 5.28　Belady 现象示例(4)

由图 5.28 可知，当进程 P 分得 4 个页面时，在执行过程中的缺页次数为 10 次。即缺页率为 10/12≈83.3％。

先进先出算法产生 Belady 现象的原因在于它根本没有考虑程序执行的动态特征。

**3. 最近最久未使用(least recently used)页面置换算法**

该算法的基本思想是：当需要淘汰某一页时，选择离当前时间最近的一段时间内最久没有使用过的页先淘汰。该算法的主要出发点是，如果某页被访问了，则它可能马上还要被访问。或者反过来说，如果某页很长时间未被访问，则它在最近一段时间也不会被访问。

要完全实现 LRU 算法是一件十分困难的事情。因为要找出最近最久未被使用的页面，就必须对每一个页面都设置有关的访问记录项，而且每一次访问都必须更新这些记录。这显然要花费巨大的系统开销。因此，在实际系统中往往使用 LRU 的近似算法。

比较常用的近似算法有以下两种：

1）最不经常使用(least frequently used，LFU)页面淘汰算法

该算法在需要淘汰某一页时，首先淘汰到当前时间为止被访问次数最少的那一页。这只要在页表中给每一页增设一个访问计数器即可实现。每当该页被访问时，访问计数器加1，而发生一次缺页中断时，则淘汰计数值最小的那一页，并将所有的计数器清零。

2）最近没有使用(NUR)页面淘汰算法

该算法在需要淘汰某一页时，从那些最近一个时期内未被访问的页中任选一页淘汰。只要在页表中增设一个访问位即可实现。当某页被访问时，访问位置 1；否则，访问位置 0。系统周期性地对所有引用位清零。当需淘汰一页时，从那些访问位为零的页中选一页进行淘汰。

**4. 理想型淘汰算法(optimal replacement algorithm，OPT)**

该算法淘汰在访问串中将来再也不出现的或是在离当前最远的位置上出现的页。这样，淘汰掉该页将不会造成因需要访问该页又立即把它调入的现象。遗憾的是，这种算法无法实现，因为它要求必须预先知道每一个进程的访问串。

## 5.4.5 存储保护

页式管理可以为内存提供两种方式的保护，一种是地址越界保护，另一种是通过页表控制对内存信息的存取操作方式以提供保护。

地址越界保护可由地址变换机构中的控制寄存器的值——页表长度和所要访问的虚地址相比较来完成。

存取控制保护的实现则是在页表中增加相应的保护位即可。

## 5.4.6 页式管理的优缺点

综上所述，页式管理具有如下优点：

(1) 由于它不要求作业或进程的程序段和数据在内存中连续存放，从而有效地解决了碎片问题。

(2) 动态页式管理提供了内存和外存统一管理的虚存实现方式，使用户可以利用的存储空间大大增加。这既提高了主存的利用率，又有利于组织多道程序执行。

其主要缺点如下：

(1) 要求有相应的硬件支持。例如地址变换机构，缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。这增加了机器成本。

(2) 增加了系统开销，例如缺页中断处理等。

（3）请求调页的算法如选择不当，有可能产生抖动现象。

（4）虽然消除了碎片，但每个作业或进程的最后一页内总有一部分空间得不到利用。如果页面较大，则这一部分的损失仍然较大。

## 5.5　段式与段页式管理

### 5.5.1　段式管理的基本思想

分区式管理和页式管理时的进程地址空间结构都是线性的，这要求对源程序进行编译、链接时，把源程序中的主程序、子程序和数据区等按线性空间的一维地址顺序排列起来。这使得不同作业或进程之间共享公用子程序和数据变得非常困难。因为用户在调用公用子程序或数据块时，往往采用指定程序名或数据块的方法。这就要求所共享的信息在逻辑上是完整的。而且，对于不同的用户进程来说，它们访问这些公用子程序或数据块的权限是不同的。因此，如果系统不能把用户给定的程序名和数据块名与这些被共享程序和数据在某个进程中的虚页对应起来，则不可能共享这些存放在内存页面中的程序和数据。另外，由于在页式管理时，一个页面中可能装有两个不同子程序段的指令代码，因此，通过页面共享来达到共享一个逻辑上完整的子程序或数据块是不可能的。

再者，从链接的角度看，分区管理和页式管理只能采用静态链接。一个大的进程可能包含数百个甚至上千个程序模块。对它们进行链接要花费大量的 CPU 时间，而实际执行时则可能只用到其中的一个子集。因此，从减少 CPU 开销和存储空间浪费的角度来看，静态链接是不合适的。

综上所述，段式存储管理是基于为用户提供一个方便灵活的程序设计环境而提出来的。段式管理的基本思想是：把程序按内容或过程（函数）关系分成段，每段有自己的名字。一个用户作业或进程所包含的段对应于一个二维线性虚拟空间，也就是一个二维虚拟存储器。段式管理程序以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换成实际的内存物理地址。和页式管理时一样，段式管理也采用只把那些经常访问的段驻留内存，而把那些在将来一段时间内不被访问的段放入外存，待需要时自动调入的方法实现二维虚拟存储器。

### 5.5.2　段式管理的实现原理

#### 1. 段式虚存空间

段式管理把一个进程的虚地址空间设计成二维结构，即段号 s 与段内相对地址 w。在页式管理中，被划分的页号按顺序编号递增排列，属一维空间，而段式管理中的段号与段号之间无顺序关系。另外，段的划分也不像页的划分那样具有相同的页长，段的长度是不固定的。每个段定义一组逻辑上完整的程序或数据。例如，一个进程中的程序和数据可被划分为主程序段、子程序段、数据段与工作区段。

每个段是一个首地址为零的、连续的一维线性空间。根据需要，段长可动态增长。对段式虚地址空间的访问包括两个部分：段名和段内地址。例如：

CALL [X] | <Y>　　　转向段名为 X 的子程序的入口点 Y

```
LOAD 1,[A]|6          将段名为 A 的数组中第 6 个元素的值读到寄存器 1 中
STORE 1,[B]|<C>       寄存器 1 的内容存入段名为 B,段中地址为 C 的单元中
```

其中的段名 X、A、B 及入口名 Y 等经编译和链接后转换成机器内部可以识别的段号和段内单元号。例如,如果[X]对应的段号为 3,<Y>对应的段内单元号为 120 的话,CALL [X]|<Y>可被编译成 CALL 3|120。

**2. 段式管理的内存分配与释放**

段式管理中以段为单位分配内存,每段分配一个连续的内存区。由于各段长度不等,所以这些存储区的大小不一。而且,同一进程所包含的各段之间不要求连续。

段式管理的内存分配与释放在作业或进程的执行过程中动态进行。首先,段式管理程序为一个进入内存准备执行的进程或作业分配部分内存,以作为该进程的工作区和放置即将执行的程序段。随着进程的执行,进程根据需要随时申请调入新段和释放老段。进程对内存区的申请和释放可分为两种情况。一种是当进程要求调入某一段时,内存中有足够的空闲区满足该段的内存要求;另一种是内存中没有足够的空闲区满足该段的内存要求。

对于第一种情况,系统要用相应的表格或数据结构来管理内存空闲区,以便对用户进程或作业的有关程序段进行内存分配和回收。事实上,可以采用和动态分区式管理相同的空闲区管理方法。即把内存各空闲区按物理地址从低到高排列或按空闲区大小从小到大或从大到小排列。与这几种空闲区自由链相对应,分区式管理时所用的几种分配算法——最先适应法、最佳适应法和最坏适应法都可用来进行空闲区分配。当然,分区式管理时用到的内存回收方法也可以在段式管理中使用。

另一种内存空闲区的分配与回收方法是在内存中没有足够的空闲区满足调入段的内存要求时使用的。这时,段式管理程序根据给定的置换算法淘汰内存中在今后一段时间内不再被 CPU 访问的段,也就是淘汰那些访问概率最低的段。

动态页式管理中的几种常用的淘汰算法都可以用来作为段式管理时的淘汰算法。例如 FIFO 置换算法、LRU 算法及其近似算法等。但是,与页式管理时每页具有相同的长度时不一样,需要调入的某段长度可能大于被淘汰的一段程序或数据的长度。这样,仅仅淘汰一段可能仍然满足不了需要调入段的内存要求。此时,就应再淘汰另外的段,直到满足需调入段的内存要求时为止。

事实上,一次调入时所需淘汰的段数与段的大小有关。如果一个作业或进程的段数较多,且段长之间的差别较大,则有可能出现调入某个大段时需淘汰好几个小段的情况。不过,在段式管理时,任何一个段的段长都不允许超过内存可用区长度,否则将会造成内存分配出错。

除了初始分配之外,段的动态分配是在 CPU 所要访问的指令和数据不在内存时产生缺段中断的情况下发生的。因此,段的淘汰或置换算法实际上是缺段中断处理过程的一部分。

缺段中断处理过程如图 5.29 所示。图中,X 代表所缺段段号。该处理程序是在 CPU 访问执行时,地址变换机构发现该段不在内存,而由硬件发出缺段中断信号后被调用的。

**3. 段式管理的地址变换**

由于段式管理只存放部分用户信息副本在内存,而大部分信息在外存中,这必然引起 CPU 访问内存时发生所要访问的段不在内存的现象。那么,CPU 如何感知到所要访问的

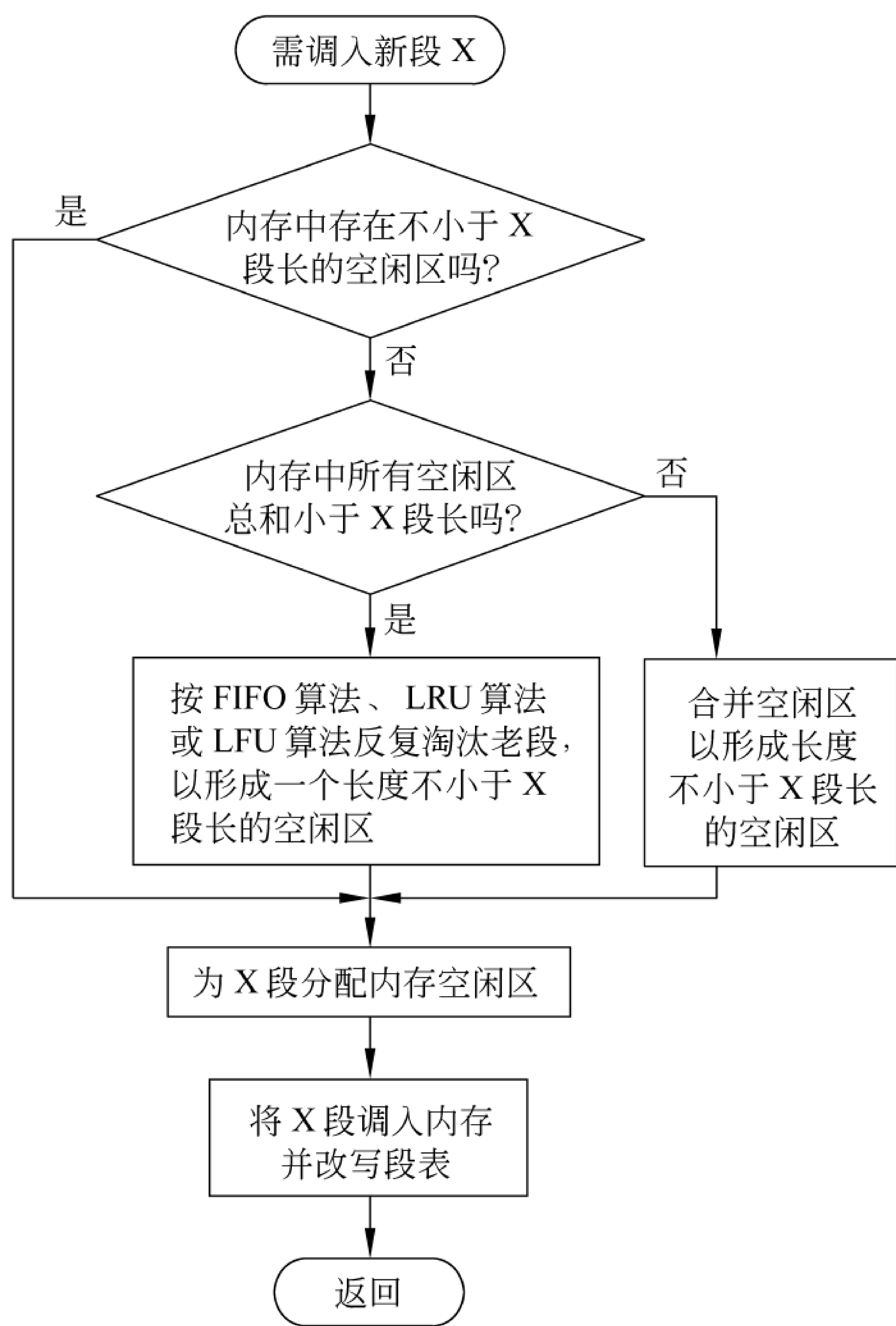



图 5.29　缺段中断处理过程

段不在内存而启动中断处理程序呢?

还有,段式虚拟地址属于一个二维的虚拟空间。一个二维空间的虚拟地址怎样变换为一个一维的线性物理地址呢?这些都由段式地址变换机构解决。

1）段表(segment mapping table)

和页式管理方案类似,段式管理程序在进行初始内存分配之前,首先根据用户要求的内存大小为一个作业或进程建立一个段表,以实现动态地址变换和缺段中断处理及存储保护等。与页式管理时一样,段式管理也是通过段表来进行内存管理的。考虑了缺段处理和段式访问控制保护后的段表如图 5.30 所示。

图中“段号”与用户指定的段名一一对应,“始址”和“长度”分别表示该段在内存或外存的物理地址与实际长度。“存取方式”是用来对该段进行存取保护的。只有处理机状态字中的存取控制位与段表中的存取方式一致时才能访问该段。“内外”是指出该段现在存储于外存还是内存中。如果该栏指出所访问段在外存的话,则发生中断。而“访问位”则是根据淘汰算法的需要而设的,这里假定淘汰算法淘汰那些访问位未被改变过的段(NUR 算法)。

2）动态地址变换

一般在内存中给出一块固定的区域放置段表。当某进程开始执行时,管理程序首先把该进程的段表始址放入段表地址寄存器。通过访问段表寄存器,管理程序得到该进程的段表始址,从而可开始访问段表。然后,由虚地址中的段号 s 为索引,查段表。若该段在内存,

| 段号 | 始址 | 长度 | 存取方式 | 内外 | 访问位 |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

图 5.30　段表

则判断其存取控制方式是否有错。如果存取控制方式正确,则从段表相应表项中查出该段在内存的起始地址,并将其和段内相对地址 w 相加,从而得到实际内存地址。

如果该段不在内存,则产生缺段中断,将 CPU 控制权交给内存分配程序。内存分配程序首先检查空闲区链,以找到足够长度的空闲区来装入所需要的段。如果内存中的可用空闲区总数小于所要求的段长时,则检查段表中的访问位,以淘汰那些访问概率低的段并将需要的段调入。段式地址变换过程如图 5.31 所示。

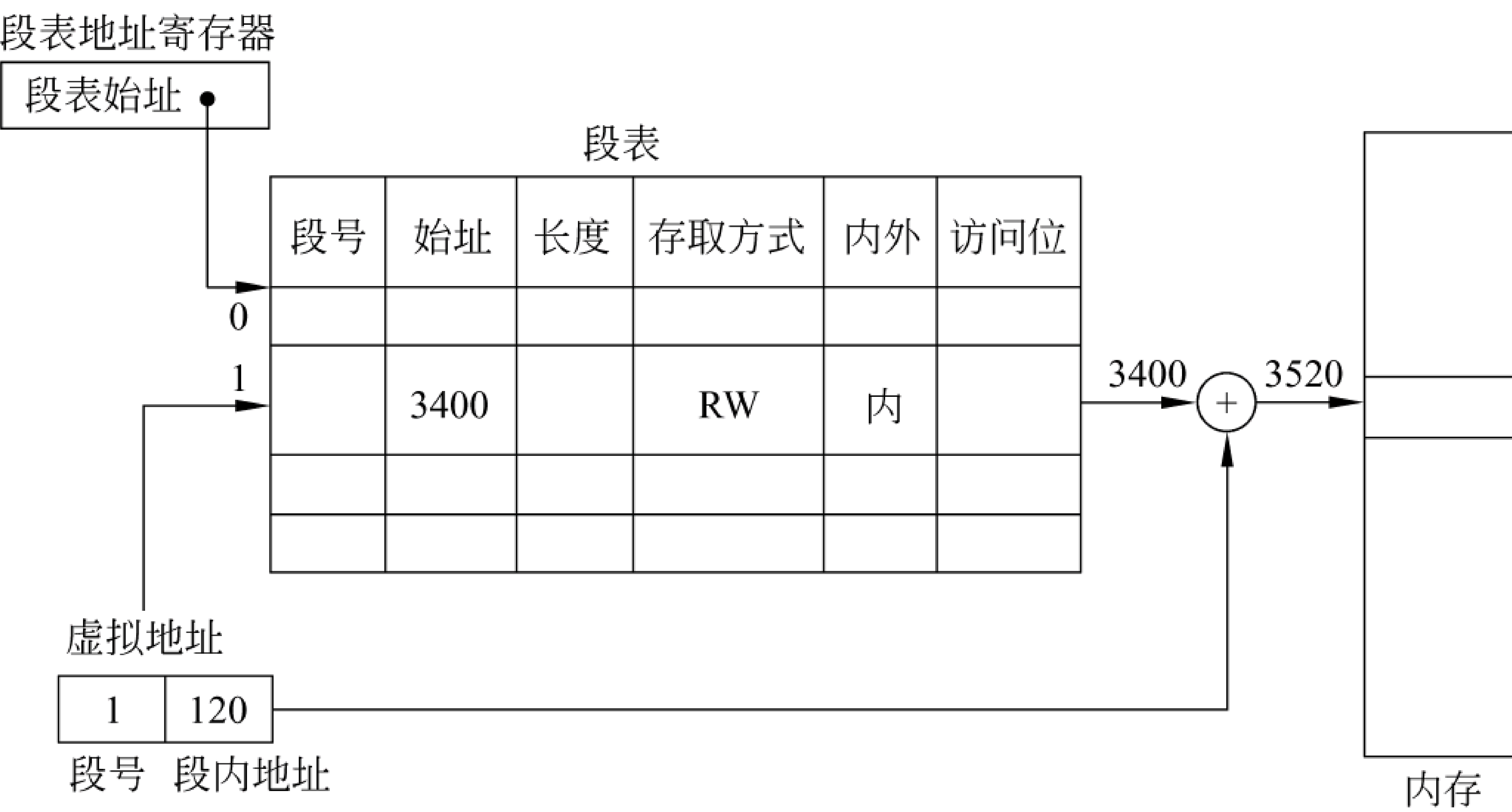



图 5.31　段式地址变换过程

与页式管理时相同,段式管理时的地址变换过程也必须经过二次以上的内存访问。即首先访问段表以计算得到待访问指令或数据的物理地址,然后才是对物理地址进行取数据或存数据操作。为了提高访问速度,页式地址变换时使用的高速联想寄存器的方法也可以用在段式地址变换中。即高速联想寄存器中存放那些经常访问的段号所对应的段表项,且高速联想寄存器中的段表和内存的段表可同时查找。如果在高速联想寄存器中找到了所需要的段,则可以大大加快地址变换速度。

**4. 段的共享与保护**

段式存储管理可以方便地实现内存信息共享和进行有效的内存保护。这是因为段是按

逻辑意义来划分的,可以按段名访问的缘故。

1) 段的共享

在多道环境下,常常有许多子程序和应用程序是被多个用户所使用的。特别是在多窗口系统、支持工具等广泛流行的今天,被共享的程序和数据的个数和体积都在急剧增加,有时往往超过用户程序长度的许多倍。这种情况下,如果每个用户进程或作业都在内存保留它们的共享程序和数据的副本,就会极大地浪费内存空间。最好的办法是内存中只保留一个副本,供多个用户使用,称为共享。图 5.32 给出了一个段式系统中共享的例子。

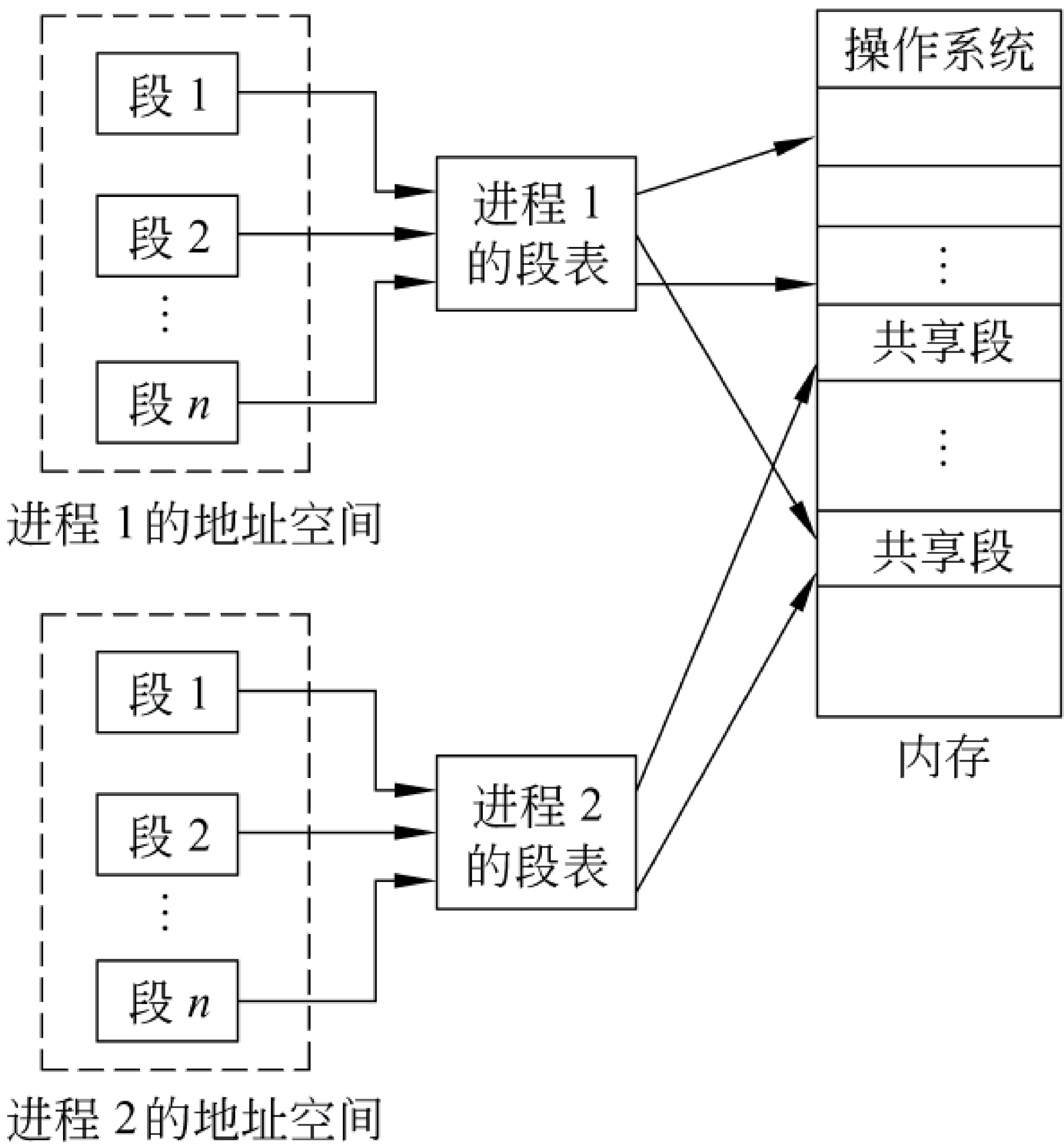



图 5.32　段式系统中共享内存副本

如图 5.32 所示,如果用户进程或作业需要共享内存中的某段程序或数据,只要用户使用相同的段名,就可在新的段表中填入已存在于内存之中的段的起始地址,并置以适当的读写控制权,就可做到共享一个逻辑上完整的内存段信息。

另外,在多道环境下,由于进程的并发执行,一段程序为多个进程共享时,有可能出现多次同时重复执行该段程序的情况(即某个进程在未执行完该段程序之前,其他并发进程又已开始执行该段程序)。这就要求在执行过程中,该段程序的指令和数据不能被修改。还有,与一个进程中的其他程序段一样,共享段有时也要被换出内存。这时,就要在段表中设立相应的共享位来判别该段是否正被某个进程调用。显然一个正在被某个进程使用或即将被某个进程使用的共享段是不应该调出内存的。

2) 段的保护

与页式管理时相同,段式管理的保护主要有两种,一种是地址越界保护法,另一种是存取方式控制保护法。关于存取方式控制保护已在前面介绍,这里不再重复。而地址越界保护则是利用段表中的段长项与虚拟地址中的段内相对地址比较进行的。若段内相对地址大于段长,系统就会产生保护中断。不过,在允许段动态增长的系统中,段内相对地址大于段长是允许的。为此,段表中设置相应的增补位以指示该段是否允许动态增长。

## 5.5.3　段式管理的优缺点

与页式管理和分区式管理比较,段式管理的长处与短处可分别总结如下。

(1) 和动态页式管理一样,段式管理也提供了内外存统一管理的虚存实现。与页式管理不同的是,段式虚存每次交换的是一段有意义的信息,而不是像页式虚存那样只交换固定大小的页,从而需要多次缺页中断才能把所需信息完整地调入内存。

(2) 在段式管理中,段长可根据需要动态增长。这对那些需要不断增加或吸收新数据的段来说,将是非常有好处的。

(3) 便于对具有完整逻辑功能的信息段进行共享。

(4) 便于实现动态链接。由于段式管理是按信息的逻辑意义来划分段,每段对应一个相应的程序模块。因此,可用段名加上段入口地址等方法在执行过程中调入相应的段进行动态链接。当然,段的动态链接需要一定的硬件支持,例如,需要链接寄存器存放被链接段的出口等。

尽管段式管理有较多的优点,但是,段式管理比其他几种方式要求有更多的硬件支持。这就提高了机器成本。另外,由于段式管理在内存空闲区管理方式上与分区式管理相同,在碎片问题以及为了消除碎片所进行的合并等问题上较分页式管理要差。再者,允许段的动态增长也会给系统管理带来一定的难度和开销。

段式管理的另一个缺点就是每个段的长度受内存可用区大小的限制。

和页式管理一样,段式管理系统在选择淘汰算法时也必须十分慎重,否则也有可能产生抖动现象。

### 5.5.4 段页式管理的基本思想

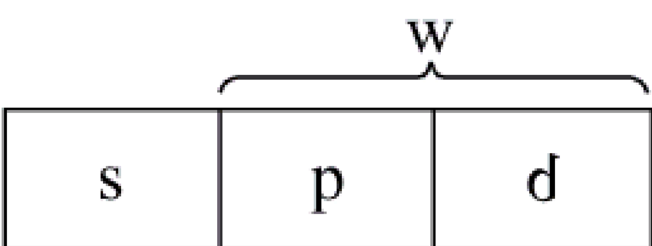
以上几种存储管理方式各有特长。段式管理为用户提供了一个二维的虚地址空间,反映了程序的逻辑结构,有利于段的动态增长、共享和内存保护等,这大大地方便了用户。而分页系统则有效地克服了碎片,提高了存储器的利用率。从存储管理的目的来讲,主要是方便用户的程序设计和提高内存的利用率。那么,把段式管理和页式管理结合起来让其互相取长补短不是更好吗? 于是,段页式管理方式便被提了出来。

不过,由于段式管理与页式管理都需要较大的系统开销,可以预计到段页式管理的开销会更大。因此,段页式管理方式一般只用在大型机系统中。近年来由于硬件发展很快,段页式管理的开销在工作站等机型上已变得可以容忍了。例如 UNIX System Ⅴ 就采用了分区加请求页式的内存管理技术。

### 5.5.5 段页式管理的实现原理

#### 1. 虚地址的构成

段页式管理时,一个进程仍然拥有一个自己的二维地址空间,这与段式管理时相同。首先,一个进程中所包含的具有独立逻辑功能的程序或数据仍被划分为段,并有各自的段号 s。这反映和继承了段式管理的特征。其次,对于段 s 中的程序或数据,则按照一定的大小将其划分为不同的页。和页式系统一样,最后不足一页的部分仍占一页。这反映了段页式管理中的页式特征。从而,段页式管理时的进程的虚拟地址空间中的虚拟地址由 3 部分组成,即段号 s、页号 p 和页内相对地址 d,如下所示:



对于这个由3部分组成的虚拟地址来说，程序员可见的仍是段号s和段内相对地址w。p和d是由地址变换机构把w的高几位解释成页号p，以及把剩下的低位解释为页内地址d而得到的。

由于虚拟空间的最小单位是页面而不是段，从而内存可用区也就被划分成为若干个大小相等的页面，且每段所拥有的程序和数据在内存中可以分开存放。分段的大小也不再受内存可用区的限制。

**2. 段表和页表**

为了实现段页式管理，系统必须为每个作业或进程建立一张段表，管理内存分配与释放、缺段处理、存储保护和地址变换等。另外，由于一个段又被划分成了若干页，每个段又必须建立一张页表，把段中的虚页变换成内存中的实际页面。显然，与页式管理时相同，页表中也要有实现缺页中断处理和页面保护等功能的表项。另外，由于在段页式管理中，页表不再是属于进程而是属于某个段，因此，段表中应有专项指出该段所对应页表的页表始址和页表长度。段页式管理中段表、页表以及内存的关系如图5.33所示，图中各表中的“其他”栏可参考段式或页式管理中的相应栏目。

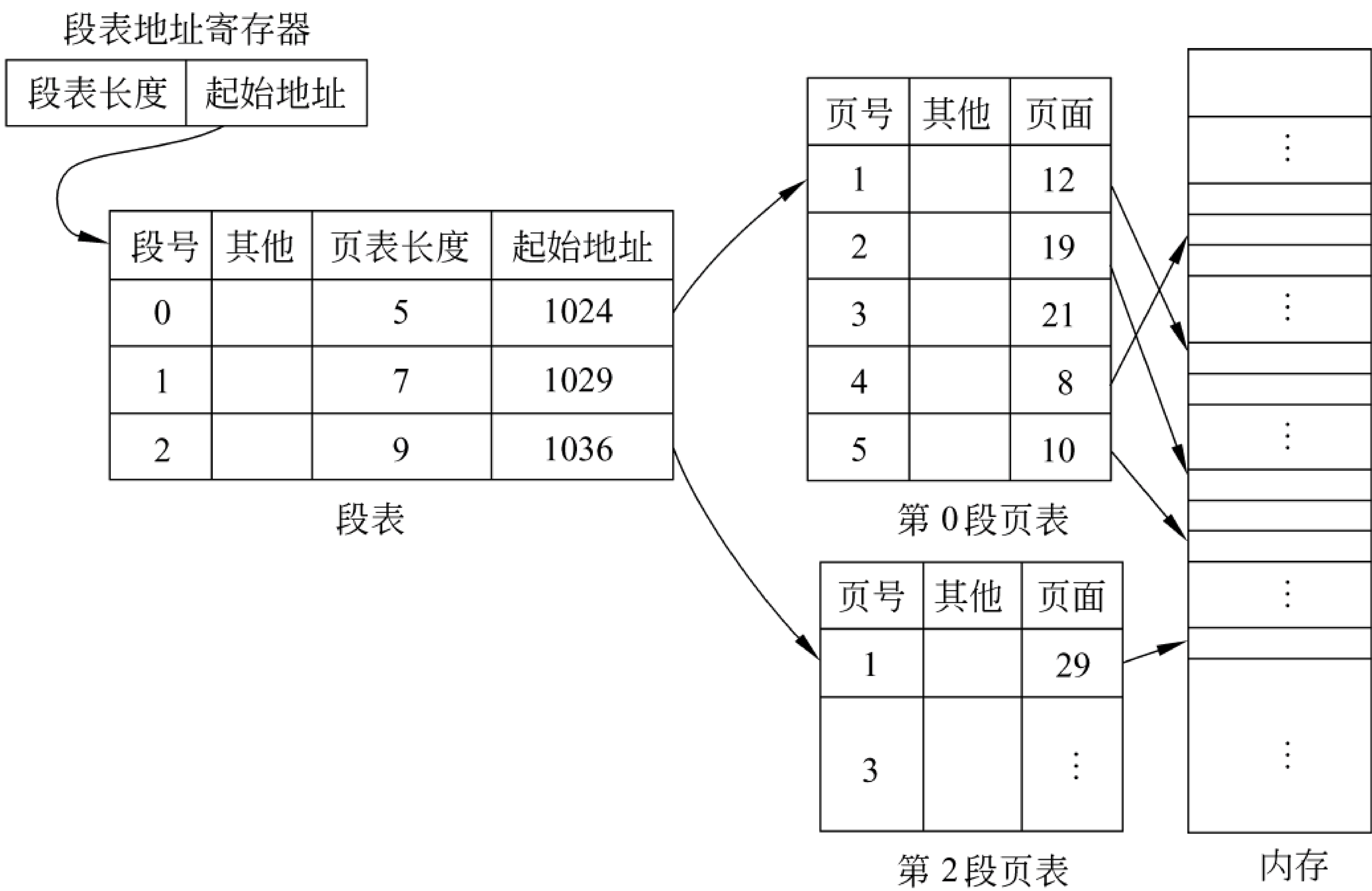



图5.33　段页式管理中段表、页表与内存的关系

**3. 动态地址变换过程**

在一般使用段页式存储管理的计算机系统中，都在内存中辟出一块固定的区域存放进程的段表和页表。因此，在段页式管理系统中，要对内存中的指令或数据进行一次存取的话，至少需要访问3次以上的内存。第一次是由段表地址寄存器得到段表始址去访问段表，由此取出对应段的页表地址。第二次则是访问页表得到所要访问的物理地址。只有在访问了段表和页表之后，第三次才能访问真正需要访问的物理单元。显然，这将使CPU的执行指令速度大大降低。

为了提高地址转换速度，设置高速联想寄存器就显得比段式管理或页式管理时更加需要。在高速联想寄存器中，存放当前最常用的段号s、页号p和对应的内存页面与其他控制

用栏目。当要访问内存空间某一单元时，可在通过段表、页表进行内存地址查找的同时，根据高速联想寄存器查找其段号和页号。如果所要访问的段或页在高速联想寄存器中，则系统不再访问内存中的段表、页表而直接把高速联想寄存器中的值与页内相对地址 d 拼接起来得到物理地址。经验表明，一个在高速联想寄存器中装有 1/10 左右的段号、页号及页面的段页式管理系统，可以通过高速联想寄存器找到 90％以上的所要访问的内存地址。

段页式管理的地址变换机构如图 5.34 所示。

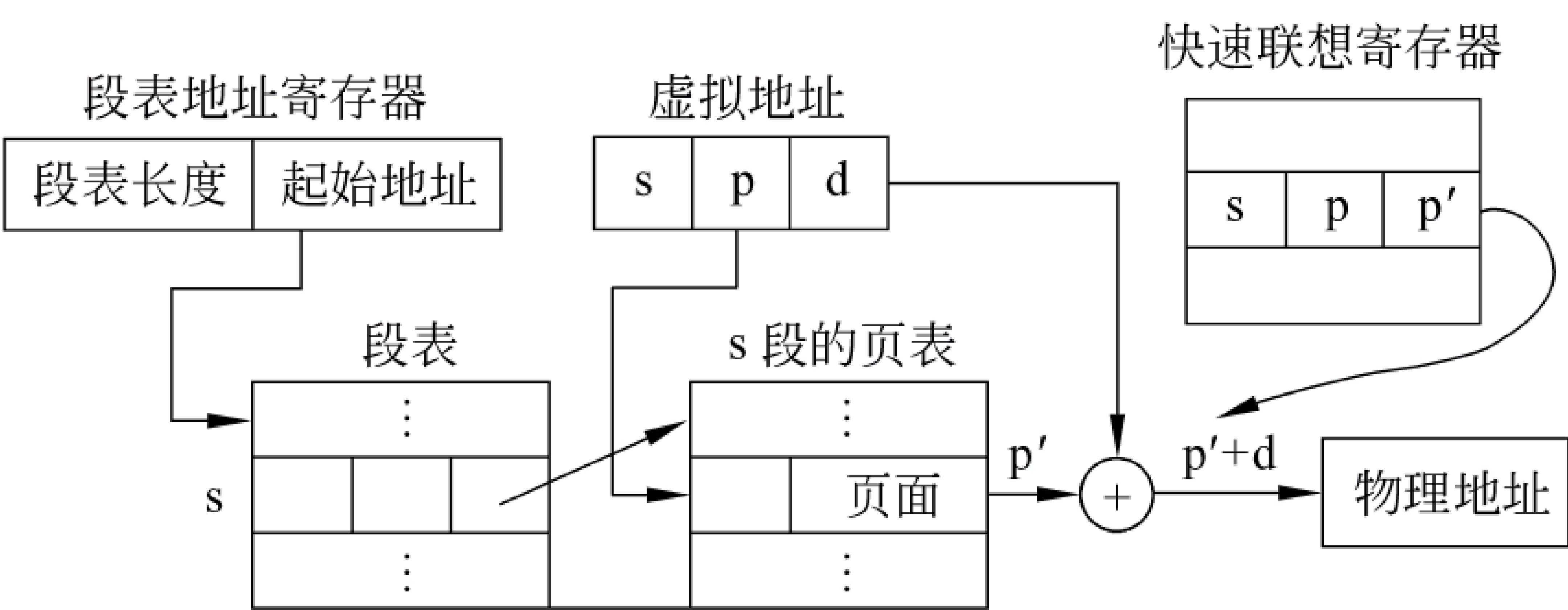



图 5.34　段页式管理的地址变换

以上简单地介绍了段页式管理中地址变换的基本原理。有关段页式管理中的存储保护和共享以及缺段或缺页中断处理等，可参照段式管理或页式管理中的方法解决。

总之，因为段页式管理是段式管理和页式管理方案结合而成的，所以具有它们二者的优点。但反过来说，由于管理软件的增加，复杂性和开销也就随之增加了。另外，需要的硬件以及占用的内存也有所增加。更重要的是，如果不采用联想寄存器的方式提高 CPU 的访内速度，将会使得执行速度大大下降。

## 5.6　局部性原理和抖动问题

动态页式管理、段式管理以及段页式管理都提供了一种将内存和外存统一管理，内存中只存放那些经常被调用和访问的程序段和数据，而进程或作业的其他部分则存放于外存中待需要时再调入内存的虚拟存储器的实现方法。然而，由于上述实现方法实质上要在内存和外存之间交换信息，因此，就要不断地启动外部设备以及相应的处理过程。一般来说，计算机系统的外部存储器与内存不同，它们具有较大的容量而访问速度并不高。而且，为了进行数据的读写而涉及的一系列处理程序(例如设备管理程序、中断处理程序等)也要耗去大量的时间。如果内存和外存之间数据交换频繁，也就是说，一个进程在执行过程中缺页率或缺段率过高，势必会造成对输入/输出设备的巨大压力和使得机器的主要开销大多用在反复调入调出数据和程序段上，从而无法完成用户所要求的工作。因此，段式、页式以及段页式虚存实现方法都要求在内存中存放一个不小于最低限度的程序段或数据，而且它们必须是那些正在被调用，或那些即将被调用的部分。这就使得内外存之间的数据交换减少到最低限度。

幸好，由模拟实验知道，在几乎所有的程序的执行中，在一段时间内，CPU 总是集中地访问程序中的某一个部分而不是随机地对程序所有部分具有平均访问概率，人们把这种现

象称为局部性原理(principle of locality)。与CPU访问该局部内的程序和数据的次数相比,该局部段的移动速率是相当慢的。这就使得前面所讨论的页式管理、段式管理以及段页式管理所实现的虚存系统成为可能。

但是,如果不能正确地将那些系统所需要的局部段放入内存,则系统的效率显然会大大降低,甚至系统无法有效地工作。

试验表明,任何程序在局部性放入时,都有一个临界值要求。当内存分配小于这个临界值时,内存和外存之间的交换频率将会急剧增加,而内存分配大于这个临界值时,再增加内存分配也不能显著减少交换次数。这个内存要求的临界值被称为工作集。图5.35说明这种情况。

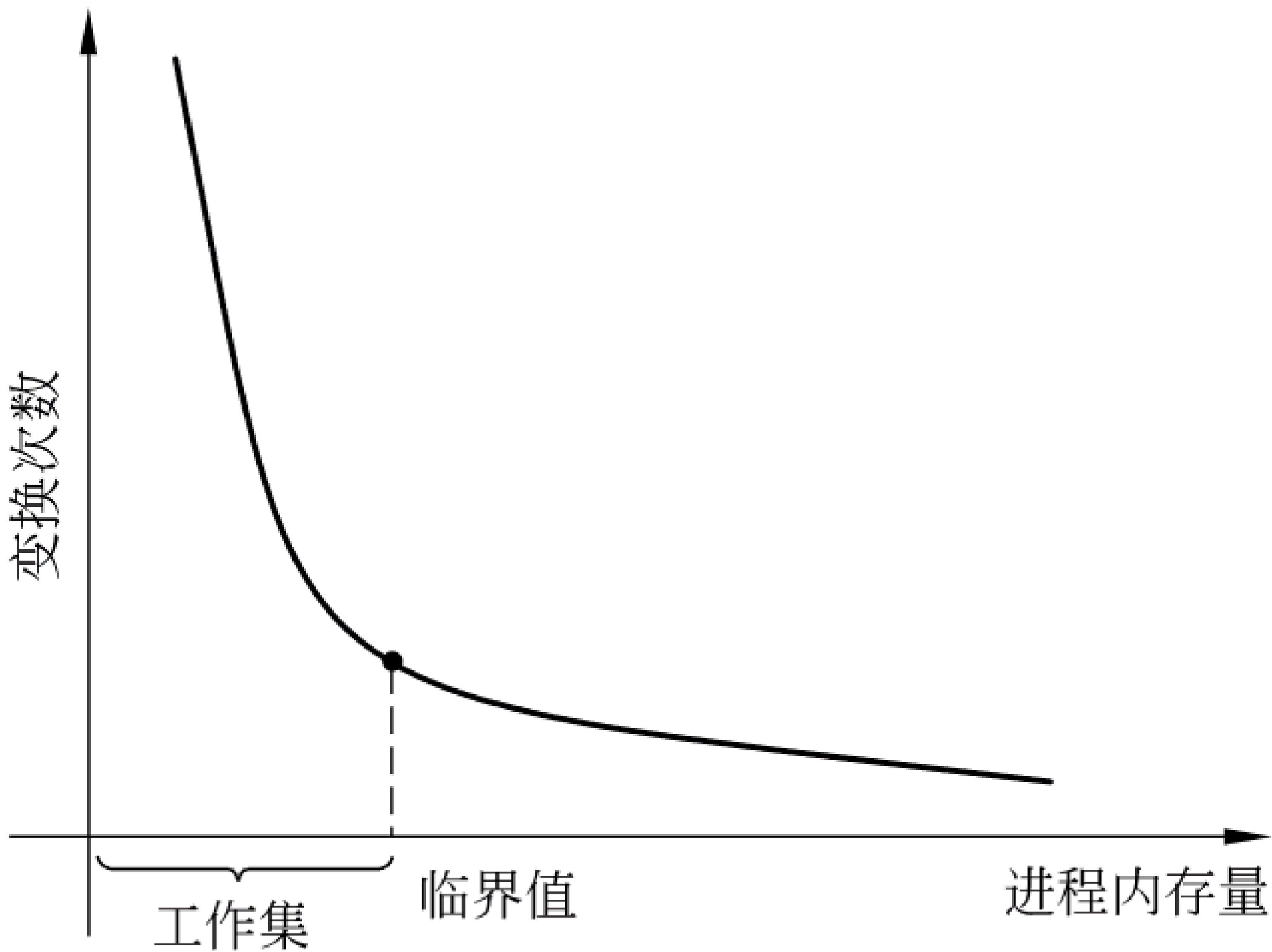



图5.35　内存与交换次数的关系

一个进程执行过程中缺页(missing page)的发生有两种可能。一种是并发进程所要求的工作集总和大于内存可提供的可用区。这时,系统将无法正常工作,因为缺乏足够的空间装入所需要的程序和数据。另一种可能性是,虽然存储管理程序为每个并发进程分配了足够的工作集,但系统无法在开始执行前选择适当的程序段和数据进入内存。这种情况下,只能依靠执行过程中,当CPU发现所要访问的指令或数据不在内存时,由硬件中断后转入中断处理程序,将所需要的程序段和数据调入。这是一种很自然的处理方法。

当给进程分配的内存小于所要求的工作集时,由于内存与外存之间交换频繁,访问外存时间和输入/输出处理时间大大增加,反而造成CPU因等待数据空转,使得整个系统性能大大下降,这就造成了系统抖动。

可以利用统计模型进一步分析工作集与抖动之间的关系。

设$r$为CPU在内存中存取一个内存单元的时间,$t$为从外存中读出一页数据所需时间,$p(s)$为CPU访问内存时,所访问的页正好不在内存的概率,这里$s$是当前进程在内存中的工作集。显然,在虚存情况下存取一个内存单元的平均时间可描述为

$$T = r + p(s) * t$$

由程序模拟可知,

$$p(s) = a\mathrm{e}^{-bs}$$

这里,$0 < a < 1 < b$,$a\mathrm{e}^{-bs} \ll r$。

另外，假定内存中各并发进程具有相同的统计特性，而且对于一个并发进程来说，只有发生缺页时才变成等待状态。这是为了简化讨论而忽略了外部设备和进程通信功能的存在。

由于访问外存一个页面的速度为 $t$，且缺页发生的概率为 $p(s)$，则在处理机访问一个内存单元的时间 $r$ 内，平均每秒引起的内外存之间页传送率为 $p(s)/r$。也就是每 $r/p(s)$ 秒需要从外存向内存传送一页。从而，对于一个在虚存范围内执行的进程，它可以处于以下 3 种可能的状态之中：

(1) $t<r/p(s)$

(2) $t>r/p(s)$

(3) $t=r/p(s)$

对于第(1)种情况，由于页传送速度大于访问外存页面的速度，因此，进程在执行过程中发生缺页的次数较少，并不经常从外存调页。

但是，在第(2)种情况时，由于内外存之间的页面传送速度已经小于访问外存页面速度，因此，进程在执行过程中发生缺页的次数已经多到外存供不应求的地步。事实上，这时的系统已处于抖动状态。

第(3)种情况是一种较理想的情况，即进程在执行过程中所需要的页数正好等于从外存可以调入的页数。此时该进程在内存中占有最佳工作集。

根据以上讨论可知，一个进程在内存中占有最佳工作集的条件是

$$p(s) = r/t$$

这里，$r$ 是 CPU 访问内存单元所需平均时间，$t$ 是访问外存一个页面所需平均时间。

因为 $p(s)$ 可表示为

$$p(s) = a\mathrm{e}^{-bs}$$

从而有

$$s = (1/b)\ln(at/r)$$

即，与内存存取速度 $r$ 相比，若外存传送速度越慢，所需工作集就越大。

当然，上面的讨论是在总结了许多近似的情况下得出的结论。事实上，由于各进程所包含的程序段多少以及选用的淘汰算法等不一样，工作集的选择也不一样。一般来说，选择工作集有静态和动态两种方法，这里不再进一步介绍。

另外，由以上讨论，可以找出解决抖动问题的几种关键办法。

抖动只有在 $t>r/p(s)$ 时才会发生。而 $p(s)$ 等于 $a\mathrm{e}^{-bs}$，是一个与工作集 $s$、参数 $a$ 和 $b$ 有关的概率值，是可以改变的。对于给定的系统来说，$t$ 和 $r$ 则是很难改变的数字。显然，解决抖动问题最关键的办法是将 $p(s)$ 减少到使 $t=r/p(s)$。这只需要采取以下两种办法之一：

(1) 增加 $s$，也就是扩大工作集；

(2) 改变参数 $a$ 和 $b$，也就是选择不同的淘汰算法以解决抖动问题。

在物理系统中，为了防止抖动的产生，在进行淘汰或置换时，一般总是把缺页进程锁住，不让其换出，而调入的页或段总是占据那些暂时得不到执行的进程所占有的内存区域，从而扩大缺页进程的工作集。UNIX System Ⅴ中就是采用的这种方法。

# 本章小结

本章介绍了各种常用的内存管理方法，它们是分区式管理、页式管理、段式管理和段页式管理。内存管理的核心问题是如何解决内存和外存的统一，以及它们之间的数据交换问题。内存和外存的统一管理使得内存的利用率得到提高，用户程序不再受内存可用区大小的限制。与此相关联，内存管理要解决内存扩充、内存的分配与释放、虚拟地址到内存物理地址的变换、内存保护与共享、内外存之间数据交换的控制等问题。

表 5.1 系统地对几种存储管理方法所提供的功能和所需硬件支持作了比较。

**表 5.1　各种存储方法比较**

| 方法<br>功能 | 单一连续区 | 分区式 | | 页式 | | 段式 | 段页式 |
|---|---|---|---|---|---|---|---|
| | | 固定分区 | 可变分区 | 静态 | 动态 | | |
| 适用环境 | 单道 | 多道 | | 多道 | | 多道 | 多道 |
| 虚拟空间 | 一维 | 一维 | | 一维 | | 二维 | 二维 |
| 重定位方式 | 静态 | 静态 | 动态 | 动态 | | 动态 | 动态 |
| 分配方式 | 静态分配连续区 | 静态或动态分配连续区 | | 静态或动态分配以页为单位的非连续区 | | 动态分配以段为单位的非连续区 | 动态分配以页为单位的非连续区 |
| 释放 | 执行完成后全部释放 | 执行完成后全部释放 | 分区释放 | 执行完成后释放 | 淘汰与执行完成后释放 | 淘汰与执行完成后释放 | 淘汰与执行完成后释放 |
| 保护 | 越界保护或没有 | 越界保护与保护键 | | 越界保护与控制权保护 | | 越界保护与控制权保护 | 越界保护与控制权保护 |
| 内存扩充 | 覆盖与交换技术 | 覆盖与交换技术 | | 覆盖与交换技术 | 外存、内存统一管理的虚存 | 外存、内存统一管理的虚存 | 外存、内存统一管理的虚存 |
| 共享 | 不能 | 不能 | | 较难 | | 方便 | 方便 |
| 硬件支持 | 保护用寄存器 | 保护用寄存器 | 保护用寄存器加重定位机构 | 地址变换机构<br>中断机构<br>保护机构 | | 段式地址变换机构，保护与中断，动态连接机构 | 段式地址变换机构，保护与中断，动态连接机构 |

# 习　题

5.1　存储管理的主要功能是什么？

5.2　什么是虚拟存储器？其特点是什么？

5.3　实现地址重定位的方法有哪几类？形式化地描述动态重定位过程。

5.4　常用的内存信息保护方法有哪几种？它们各自的特点是什么？

5.5　如果把 DOS 的执行模式改为保护模式，起码应作怎样的修改？

5.6　动态分区式管理的常用内存分配算法有哪几种？比较它们各自的优缺点。

5.7　5.2 节讨论的分区式管理可以实现虚存吗？如果不能，需怎样修改？试设计一个分区式管理实现虚存的程序流程图。如果能，试说明理由。

5.8　什么是覆盖？什么是交换？覆盖和交换的区别是什么？

5.9　什么是页式管理？静态页式管理可以实现虚存吗？

5.10　什么是请求页式管理？试设计和描述一个请求页式管理时的内存页面分配和回收算法(包括缺页处理部分)。

5.11　请求页式管理中有哪几种常用的页面置换算法？试比较它们的优缺点。

5.12　什么是 Belady 现象？试找出一个 Belady 现象的例子。

5.13　描述一个包括页面分配与回收、页面置换和存储保护的请求页式存储管理系统。

5.14　什么是段式管理？它与页式管理有何区别？

5.15　段式管理可以实现虚存吗？如果可以，简述实现方法。

5.16　为什么要提出段页式管理？它与段式管理及页式管理有何区别？

5.17　为什么说段页式管理时的虚地址仍是二维的？

5.18　段页式管理的主要缺点是什么？有什么改进办法？

5.19　什么是局部性原理？什么是抖动？你有什么办法减少系统的抖动现象？

# 第 6 章　进程与存储管理示例

本章以 Linux 2.6 内核版本为例，介绍 Linux 的进程和存储管理方法。

## 6.1　Linux 进程和存储管理简介

Linux 系统的核心部分从整体上可以分为两大部分，即“静”的文件系统和“动”的进程控制系统。文件系统主要用来存放、管理那些暂时不被处理机执行的程序和数据，它为程序和数据文件分配存储空间，控制文件存取和为用户检索信息。而进程控制系统则负责为将要执行的程序和数据文件分配内存空间，并负责进程调度，控制并发进程的执行速度，分配必要的资源，以及负责进程通信和内存管理等。

Linux 的进程控制系统与文件系统之间通过数据结构和函数调用来互相作用。Linux 的文件系统以及文件系统与进程控制系统的接口部分将在第 10 章中讲述。这里先介绍 Linux 的进程控制系统部分。

Linux 系统把一个程序看作是一个可执行文件，而把一个进程看作是程序的执行或执行中的程序实例。但是，从静态的观点看，CPU 把进程解释为由一组机器指令、数据和堆栈结构组成的集合及其上下文。由于调度并不一定是在每个进程执行完毕时发生，因此，系统内可以同时有多个进程在执行。而且，若干个进程可以同时调用同一个程序。

和其他所有操作系统一样，Linux 操作系统只有在其内核装入内存后才能开始运行。

为了使操作系统内核能在每次开机时顺利地装入内存，用户必须事先把 Linux 操作系统的执行代码以文件方式存储在计算机硬盘设备中，并对计算机系统中的相应资源，例如引导程序、交换区等进行初始化。这一过程被称为操作系统的安装过程。Linux 的各种发布版本都有自己的安装程序。用户只要按照安装程序的提示和说明一般都能顺利地进行系统安装。因此，本章的进程运行和存储管理都是假定在一个已完全安装好的操作系统基础上进行的。

在一个已安装好 Linux 系统的计算机中，启动电源意味着系统引导程序开始系统自举，将保存在外存硬盘中的操作系统核心加载到内存，然后进行 Linux 核心的初始化。这一过程是一个设置和初始化各种数据结构与表格、初始化 Linux 核心的各个子系统的过程。从进程创建的角度来看，这一过程首先建立了 Linux 系统中唯一一个静态建立的进程（在 Linux 系统中把该进程称为 0# 进程，或 idle 进程），之后建立了控制终端进程及运行 Shell 进程（用户交互进程）的 init 进程（有时也称其为 1# 进程）。系统在建立了 Shell 进程之后，将出现相应的提示符，等待用户输入命令来执行和处理用户应用程序。

Linux 系统在初始过程中的运行如图 6.1 所示。在图 6.1 中，当系统创建了 init 进程之后，init 进程将会调用相应的终端管理进程，为 Linux 系统的不同终端创建相应的终端管理进程和 Shell 进程，从而使每个终端上的用户都可以在 Shell 的管理下交互使用 Linux 系统。

Shell 程序将为用户提供解释执行用户命令的交互工具。随着用户命令，例如 cp 等的输入，系统将建立一个执行该命令的用户进程。Shell 进程则会进入等待队列，以便用户进程执行。在用户进程执行结束后，Shell 进程又恢复执行，显示提示符并等待用户的下一条命令输入。Linux 系统的所有进程都是在上述执行过程中产生和消亡。

由图 6.1 可以看到，终端管理进程与 Shell 进程是 init 进程的子进程，而其他的用户进程则是由 Shell 进程创建的。所以在 Linux 系统中，除了 0# 进程以外的所有进程都是由 init 进程衍生出来的，从而人们也称 init 进程是所有用户进程的祖先。

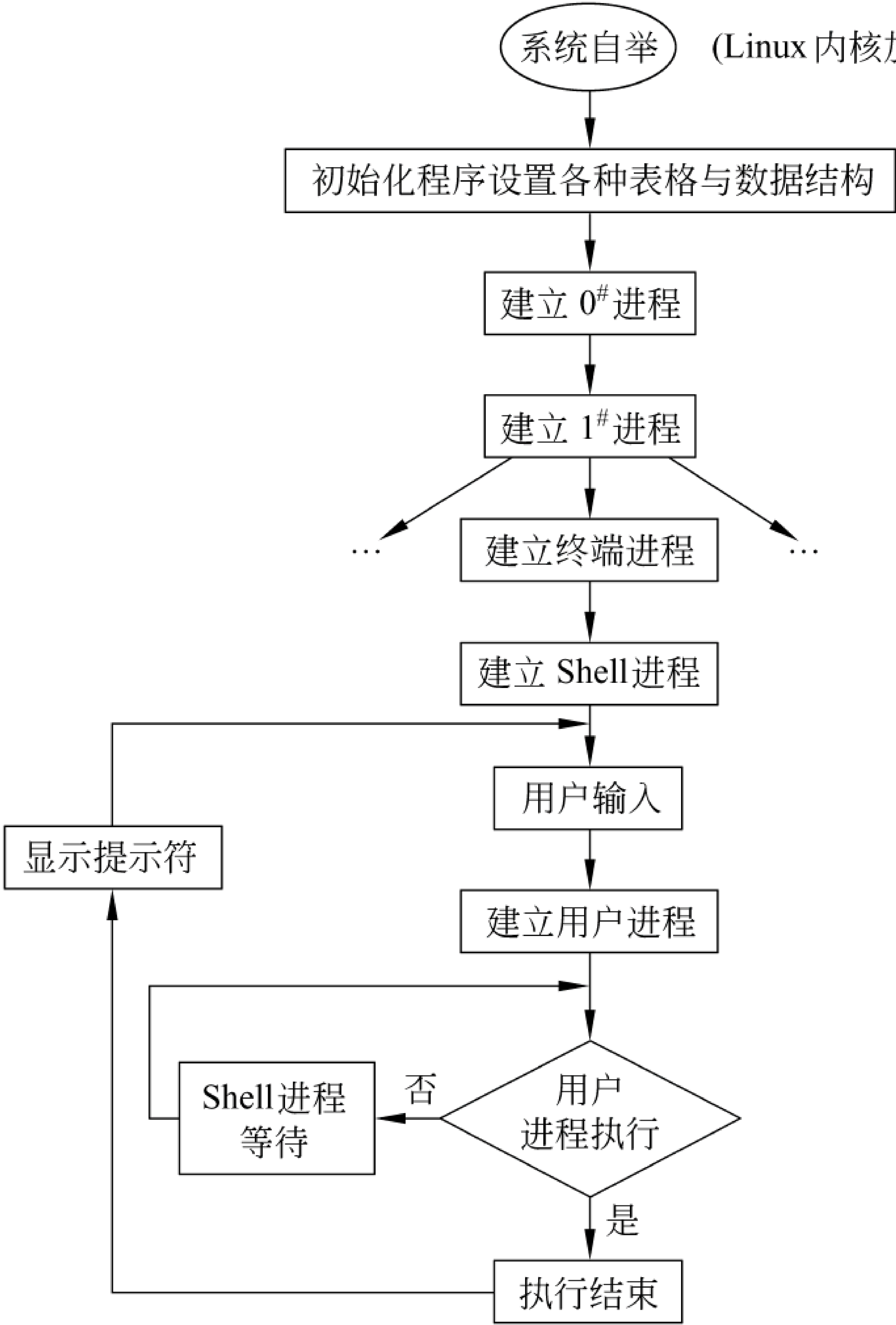



图 6.1　Linux 系统中各进程的关系

Linux 操作系统的 0# 进程在核心态下运行，而 init 进程以及由 init 进程衍生的其他进程都可在用户态和核心态两种执行模式下执行。下面先介绍用户态和核心态。

为了在操作系统和用户进程之间进行隔离，处理器通常提供两种不同权限的执行模式：核心态或用户态。对于 Intel 80x86 处理器，实际上提供了 4 个不同权限的执行模式，但是 Linux 系统主要使用其中两种模式，所以不失普遍性，下面只描述为两种模式：核心态和用户态。

两种执行状态之间的主要区别是：用户态下的进程能存取它们自己的指令与数据，但不能存取核心指令和数据；而核心态下的进程能存取核心和用户地址。另外某些机器的指令是特权指令（例如输入输出指令），在用户态下执行会引起错误，只能在核心态下执行。操作系统运行在核心态，用户进程通常运行在用户态，只有当用户进程需要请求操作系统的服

务时，才通过系统调用切换到核心态，操作系统完成服务后再次将用户进程切换回用户态。这样就达到了在操作系统和用户进程之间进行安全隔离的目的。

在不同的执行模式下执行时，同一进程使用不同的堆栈，分别称为核心态堆栈和用户态堆栈。在进程切换到不同的执行模式的时候，由 Linux 操作系统负责为进程切换到相应的堆栈。

Linux 系统中进程通过请求操作系统服务进入核心态的机制称为系统调用。具体的实现和硬件平台的体系结构相关，例如在 80x86 系统中，用户进程通过 int 0x80 指令请求系统调用，系统调用完成后通过 iret 指令返回到用户态。它们之间的关系如图 6.2 所示。

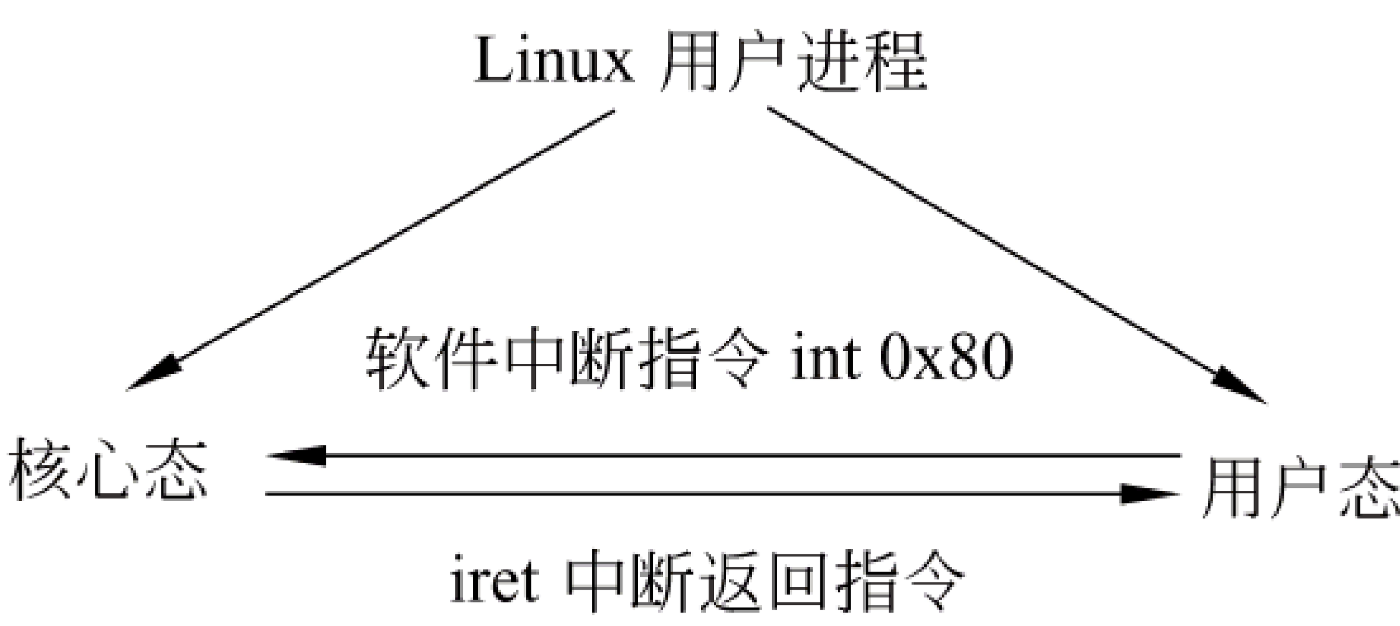



图 6.2　Linux 进程的核心态与用户态之间的转换

上面说过 0# 进程是只在核心态下执行的进程。在 Linux 中这样的进程称为核心线程(kernel thread)。核心线程除了 0# 进程外还有 kblockd、ksoftirqd 和 kswapd 等，它们通常用于处理一些周期性的事务。它们的代码在核心态运行，所以和核心拥有相同的权限。和核心使用同样的地址空间，所以没有用户空间上下文，因而核心线程的调度开销也比较小。

Linux 的进程控制系统在逻辑上由 4 个模块组成：与文件系统的接口部分；进程本身的控制部分，包括进程的创建、进程调度和进程的撤销等；进程间控制部分，包括进程间的互斥、同步和通信等；以及存储管理部分。

调度模块的作用是分配 CPU。显然，在进行资源分配特别是 CPU 分配时要按照一种既公平合理又有很高效率的分配原则。在 Linux 系统中，这个调度原则就是按照进程的优先级，每次调度具有最高优先级的进程去占有处理机。每一个进程，从它被创建的那一时刻起，就具有了一个随时间动态变化的优先级。在 Linux 中，调度过程 schedule()和时钟中断都会修改这个动态的优先级。

Linux 中引起进程调度的情况有下面几种：

(1) 当前执行进程申请系统资源未得到满足，从而自己调用 sleep 过程，放弃处理机而进入睡眠状态。

(2) 为了与其他并发进程保持同步调用了 wait 过程等，从而主动放弃了处理机而进入睡眠状态。

(3) 当某个进程被唤醒，并发现它的优先级高于当前进程的优先级时，以及当前进程的时间片用完时，系统设置一个名叫 need_resched 的调度标志，但是，系统并不在 need_resched 标志刚被设置时就开始调度，而要等到系统在核心态下的程序执行完毕，由核心态转入用户态时，也就是在中断陷入总控处理程序结束之前的瞬间，检查 need_resched 标志并进行调度。

(4) 当前进程调用 exit，自我终止时。

有关进程控制和调度，将在 6.3 节和 6.4 节中做更进一步的说明。

进程通信是 Linux 的一个重要组成部分。进程通信既包括用来控制各并发进程执行速度和资源共享与竞争的低级通信，也包括进程之间大量传递信息的高级通信。

Linux 系统在核心态下执行时，系统进程可以使用下列同步机制：信号量、自旋锁、关闭中断和使用原子操作等。用户进程不能使用这些核心态的同步方法、用户进程之间要实现同步通信有两种办法。一种办法是利用系统核心提供的软中断信号进行通信。Linux 为用户使用软中断通信提供了相应的系统调用。另一种办法是调用系统调用 wait 或 sleep 使得当前执行进程进入等待状态，直到所等待事件发生时由核心唤醒或睡眠到一定时间后自动唤醒。用户进程之间实现同步的高级办法是信号量机制，其概念和原理与用户态的 UNIX System Ⅴ的 IPC 机制信号量一样。

System Ⅴ中的 IPC(Inter-Process Communication)模块为进程间的大量信息传送提供了众多的系统调用。

除了上述通信手段之外，Linux 还提供了称为管道(pipe)和命名管道(fifo)的信息传送手段。它以文件的方式，实现同一主机内的进程之间批量数据的先进先出方式的无格式字符流传送。6.5 节中将较为系统地介绍 Linux 的进程通信。

存储管理模块控制存储分配。任何一个待执行进程，如果不能在内存中占据必需的容量，就不能执行。换句话说，CPU 绝对不会执行一个全部内容都在外存的进程。然而，内存是一种有限而又昂贵的资源，它不可能容纳系统中全部活动进程。存储管理系统必须决定把哪一个进程的哪一部分留在内存中，而把哪一部分放在外存。也就是说，存储管理部分管理进程在内存和外存之间的信息转移，以便所有进程都得到公平执行的机会。6.6 节中将介绍 Linux 的存储管理策略——请求调页。

下面先介绍 Linux 系统的进程结构。

## 6.2 Linux 进程结构

6.1 节简单地介绍了进程与存储管理的一些高层次特性。本节进一步介绍 Linux 进程的静态构成，定义进程上下文及其状态转换等。

### 6.2.1 进程的概念

在 Linux 系统中，进程被赋予了下述特定的含义和特性：

(1) 一个进程是对一个程序的执行。

(2) 一个进程的存在意味着存在一个 task_struct 结构，它包含着相应的进程控制信息。

(3) 一个进程可以生成或消灭其子进程。

(4) 一个进程是获得和释放各种系统资源的基本单位。

上述第(1)点是反映进程动态特性的，而第(2)点又反映了进程的静态特性，第(3)点与第(4)点反映了 Linux 系统的进程之间的关系以及 Linux 没有作业概念的特性。由第 3 章可知，一个进程的静态描述是由 3 部分组成的，即进程状态控制块(栈段)，进程的程序文本(正文)段以及进程的数据段。在第 3 章中，把这 3 部分统称为进程上下文，而进程的动态特性则定义为在进程上下文中的执行。

下面介绍 Linux 的进程控制块。

Linux 的 task_struct 结构相当于第 3 章中介绍的进程控制块(PCB)。与 UNIX 将进程控制块分割为长驻内存的 proc 和可交换到外存的 user 两个结构不同,Linux 只使用 task_struct 这样一个独立的数据结构来表示进程控制块。task_struct 是一个相当庞大的数据结构,下面只讨论其中重要的部分:

(1) 标识进程的状态用的状态位。Linux 共有 6 种基本状态。有关这些状态的转换和条件,将在 6.2.4 节介绍。

(2) 进程标识号(PID),用来唯一地标识一个进程。

(3) 描述进程家族关系、组成员关系的一些指针,用来说明进程相互间的关系。

(4) 若干用户标识号(ID),包括用户 ID(UID)、组 ID(GID)等。这些用户标识号指出该进程属于哪一组用户,进而表明了该进程具有何种权限,例如可以访问哪些文件。

(5) 调度参数,包括优先数、调度策略、进程所处的就绪队列和时间片等。

(6) 软中断信号项,记录和软中断信号相关的信息。

(7) 中断及软中断处理的有关参数,依靠这些参数,进程对所收到的软中断信号作出不同的反应。

(8) 各种计时项,给出进程执行时间和系统资源的利用情况。这些信息用来为进程记账、计算调度优先权以及发送计时信号等。

(9) 进程地址空间和内存关系有关的信息,这是一个 mm_struct 类型的成员,它描述了进程线性区域、进程数据段、正文段、堆段起始位置、长度和进程页表指针。

(10) 与文件系统有关的若干项,这是一个 fs_struct 类型的成员。该结构描述了进程的当前目录和当前根、进程的文件系统环境、文件设置许可权方式以及字段的屏蔽模式等。

(11) 用户文件描述符表,记录该进程已打开的文件。

(12) 进程消亡时的返回值和终止信号,父进程通过检查这些参数来了解进程的运行状况。

(13) 与上下文切换、现场保护有关的各项,它们保存各种通用寄存器和程序计数器的当前值、处理机状态字、用户栈指针和进程的整个核心栈。

(14) 资源限制有关的各项,它们保存进程对资源访问的上限。

由以上 task_struct 结构的组成可知,task_struct 结构中存放的是系统感知进程存在所必需的数据和信息,以及进程执行时所必需的各种控制数据和信息。

### 6.2.2 进程的虚拟地址结构

由于 Linux 进程的虚拟地址结构依赖于硬件,因此,如果不作特别说明,本文默认那些与硬件有关的部分都是依赖于 Intel 80x86 的。80x86 平台中,每个进程拥有一个 4GB 的虚拟空间。其中 0～3GB 的地址空间由用户进程使用,用户进程可以对其直接访问。3～4GB 的地址空间称为核心地址空间,在所有的进程中共享,存放核心的正文和数据,只被核心使用,用户进程不能直接访问。

Linux 将用户进程的所有地址空间有关的信息保存在一个名为 mm_struct 的数据结构中,该数据结构自身(实际上是它的指针)则保存在进程描述符中,这个在前面介绍 Linux 的进程描述符的时候已经提到过。

Linux 的进程由逻辑段组成的,例如有存放状态控制块的栈段、存放 CPU 执行指令集合的正文段以及被执行指令所访问的数据段。相应的 Linux 中,一个进程的虚拟地址空间被分成若干个虚拟区域来存放上述的逻辑段。区是进程虚拟地址空间上的一段连续区域,它是被共享、保护以及进行内存分配和地址变换的独立实体。正文、数据和栈分别存放于各自的区中。在 Linux 中虚拟区域被命名为 vm_area,在核心代码中通常简写为 VMA。

为了管理每个进程中的区,系统设有一个称为 vm_area_struct 的数据类型,进程的每个区都对应一个 vm_area_struct 结构,它主要包括下列内容:

(1) 区的标志位,指明该区的类型以及是否被锁住,是否可共享等属性。缺页处理程序会根据地址所在区的标志位查找缺页原因,并做相应处理。

(2) 区的起始地址和结束地址。

(3) 共享区域指针,给出共享此区的 vm_area_struct 链表。

(4) 文件系统指针,指向外存中与该区对应的数据文件。

(5) 此区域的操作函数指针。

在系统创建新进程时,核心将从父进程复制相应的表项给所创建的进程。

这里,要强调的一点是:对于一个进程,它所有的区的地址范围绝对不会重叠,两个区的虚拟地址不一定连续,而进程的虚拟地址在各区之内是连续的。

为了加快对区域的查找和插入删除操作,Linux 使用 AVL 平衡二叉树来组织和管理区域。

对于用户进程,它可以通过系统调用 mmap()请求创建一个虚拟区域,并通过 munmap()系统调用加以释放。

在虚拟区域的讨论中,大家也可能注意到一点,即 Linux 中的区和段页式管理中的段非常相像。所不同的是,段页式管理中的虚拟地址空间是二维的,而 Linux 的各个进程的分区虚拟地址仍是一维的。

### 6.2.3 进程上下文

在第 3 章中,已部分地介绍了 Linux 进程上下文的概念。Linux 的进程上下文是由正文段,也就是 CPU 执行指令的集合、核心数据结构、有关寄存器的内容与数据段组成。

**1. 进程上下文的基本结构**

进程上下文的各个部分按照一定的规则分布在进程虚拟空间的不同位置上。对于不同的机器和硬件结构,进程上下文的分布规则不同。例如,在 80x86 上,其虚拟地址空间划分为进程空间和系统空间两大部分。其寻址范围为 $2^{32}$ 个单元,即 4GB。其中,虚拟空间的 0～3GB 是进程虚拟空间,其余为所有进程共享的系统空间,进程虚拟空间又分为多个虚拟区域,分别保存程序正文、数据、堆和用户栈。其中堆空间由低向高动态延伸,而栈空间由高向低动态延伸。

在图 6.3 中,核心态栈(kernel stack)是该进程执行核心程序时使用的,如上面所述,核心栈空间(实际上是指向它的指针)保存在进程的进程描述符中。该栈中装有进程调用核心函数时用到的有关参数等,另外,还包括系统调用的调用序列。设置核心栈的目的是使进程在执行了不同调用顺序的核心函数之后,仍能返回到原来的用户态下执行。核心栈具有多个层次,其中每层可保留一次调用或中断处理后返回被中断程序处继续执行所必需的有关

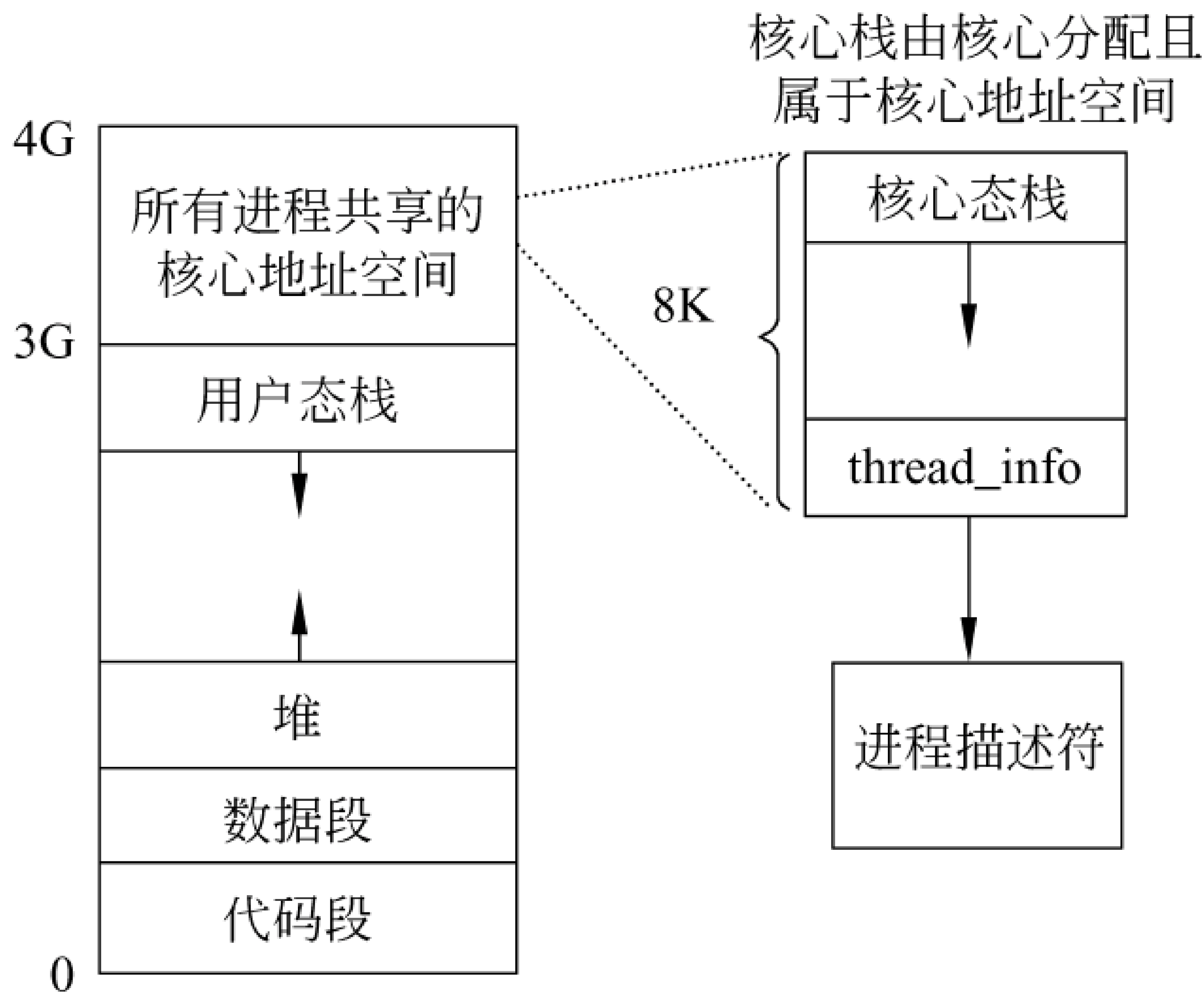



图 6.3　进程空间结构

寄存器的值。

用户栈含有在用户态下执行时函数调用的参数、局部变量及其他数据。图 6.4 给出了执行 copy 程序时用户栈和核心栈的变化例。图 6.4 的左侧描述了当由 main(argc,argv)过程调用 copy(old,new),过程 copy(old,new)更进一步调用库函数 write()来调用 write 的内部结构。系统调用使用专门的陷阱指令 trap,执行 trap 指令将产生一个中断,使得进程的执行模式由用户态转换为核心态。

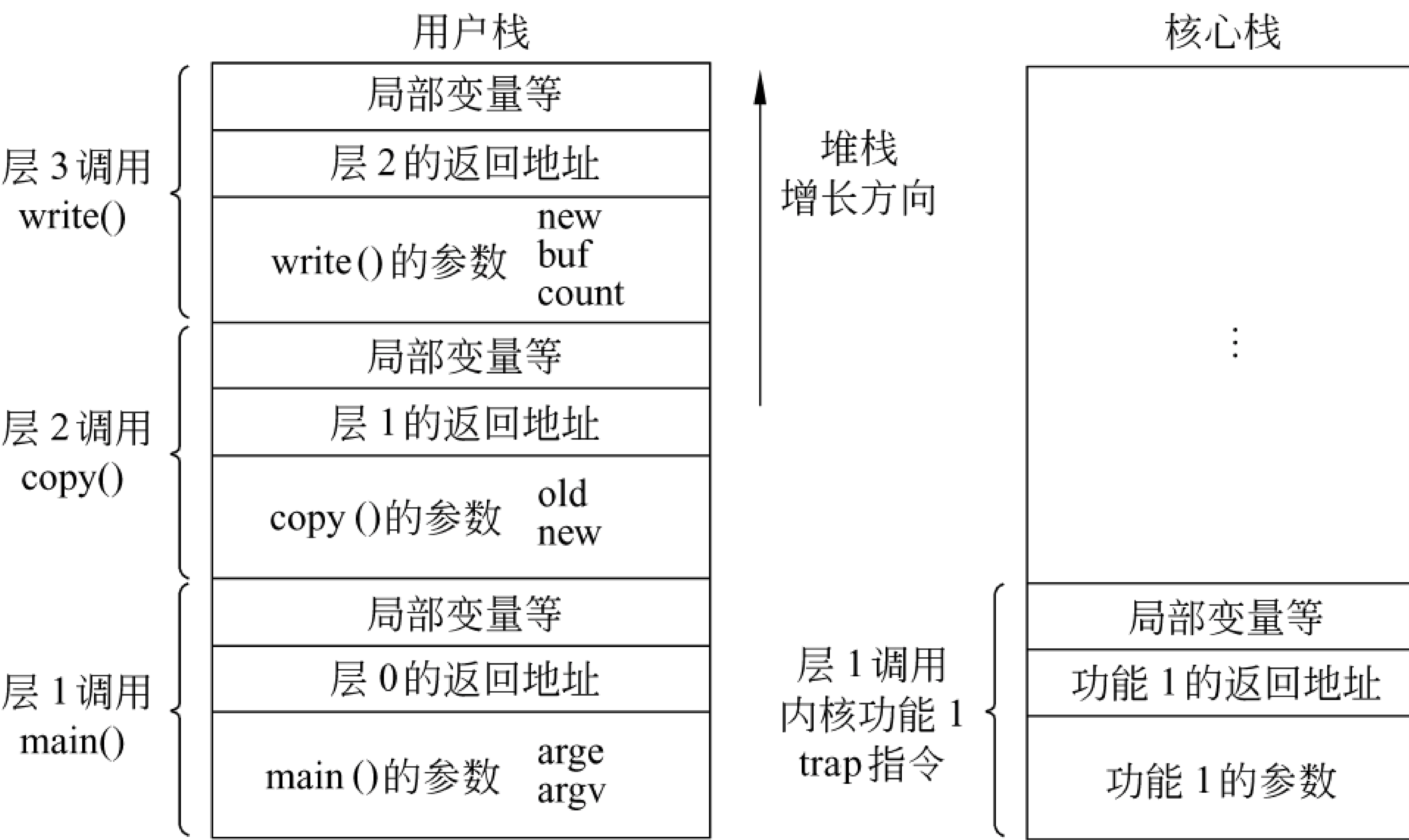



图 6.4　执行 copy 程序时用户栈和核心栈的变化

然后,用户态执行转变为核心程序执行,并使用核心栈。

进程空间的用户进程正文段、数据段、用户栈以及有关的专用代码和数据是以进程为单位独立的,它们根据需要换入换出内存。

**2. 进程上下文的组成部分**

进程上下文由 task_struct 结构结构、用户栈和核心栈的内容、用户地址空间的正文段、数据段、硬件寄存器的内容以及页表等组成。

task_struct 结构在前文已经进行过介绍。

1）寄存器内容

进程上下文所包含的寄存器内容如下：

（1）程序计数器 IP 的内容，指出 CPU 将要执行的下一条指令的虚拟地址。

（2）处理机状态寄存器的内容，称为处理机状态字。它给出机器与该进程相关联的硬件状态。例如运算结果的标志位、中断屏蔽位、IO 特权位等。

（3）栈指针，指向栈中下一项的当前地址。至于指针是指向核心栈还是用户栈，则由 CPU 执行方式确定。

（4）通用寄存器，用来存放进程在执行期间所产生的中间结果或参数。例如，通用寄存器 EAX 就是在用户进程与系统进程之间传递参数和返回值时用的。

2）页表

页表也是进程上下文的内容之一。它定义了进程各部分从虚拟地址到物理地址的映射。每个进程都有自己独立的页表，在进程被调度的时候，调度过程会负责将当前页表切换为进程的页表。正是页表使每个进程拥有自己的虚拟地址空间。

3）进程正文段和数据段

进程正文段、数据段和用户栈等一起占据进程的虚拟地址空间部分，由页表组成的分页地址变换机构将其虚拟地址变换为内存物理地址。由于 Linux 采用请求页式虚存，因此，进程虚拟地址空间的许多部分不是总驻留在内存中，这一点与 UNIX 的早期版本有区别，但它们仍是进程上下文的一部分。

### 6.2.4 进程的状态和状态转换

正如在第 3 章中所指出的那样，一个进程的生命期是由一组状态来刻画的。这些状态是进程 task_struct 结构的一部分。

Linux 系统中进程共有 6 种基本状态。

（1）TASK_RUNNING：运行状态。进程处在执行或就绪状态，表示正在占用 CPU，或者在就绪队列中等待调度，只要调度到它，就可投入执行。Linux 系统中没有就绪队列，已就绪进程和当前运行进程的状态都是 TASK_RUNNING。

（2）TASK_INTERRUPTIBLE：可中断状态。进程正在睡眠，但是可以被软中断信号唤醒。当收到信号时，进程会从等待队列移动到运行状态执行信号处理程序，如果优先级高于正在运行的进程，将抢占直接执行。

（3）TASK_UNINTERRUPTIBLE：不可中断状态。进程正在睡眠，且不可以被软中断信号唤醒。

（4）TASK_STOPPED：暂停状态。表示进程的执行被暂停，当一个进程收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 软中断信号后会进入这个状态。在调试期间，进程收到任何信号，也会停止运行。

（5）TASK_TRACE：表示进程被 debugger 等进程监视。

（6）TASK_ZOMBIE：进程执行了系统调用 exit 后，进入僵死状态。

图 6.5 反映了一个进程从被创建到被释放的整个生命周期内的变化过程。

需要说明的是，如前面所描述的，在 Linux 进程描述符的状态位中并不区分就绪状态和运行状态，它们统一称作 TASK_RUNNING 状态，而图中的睡眠状态在 Linux 的状态位中则分

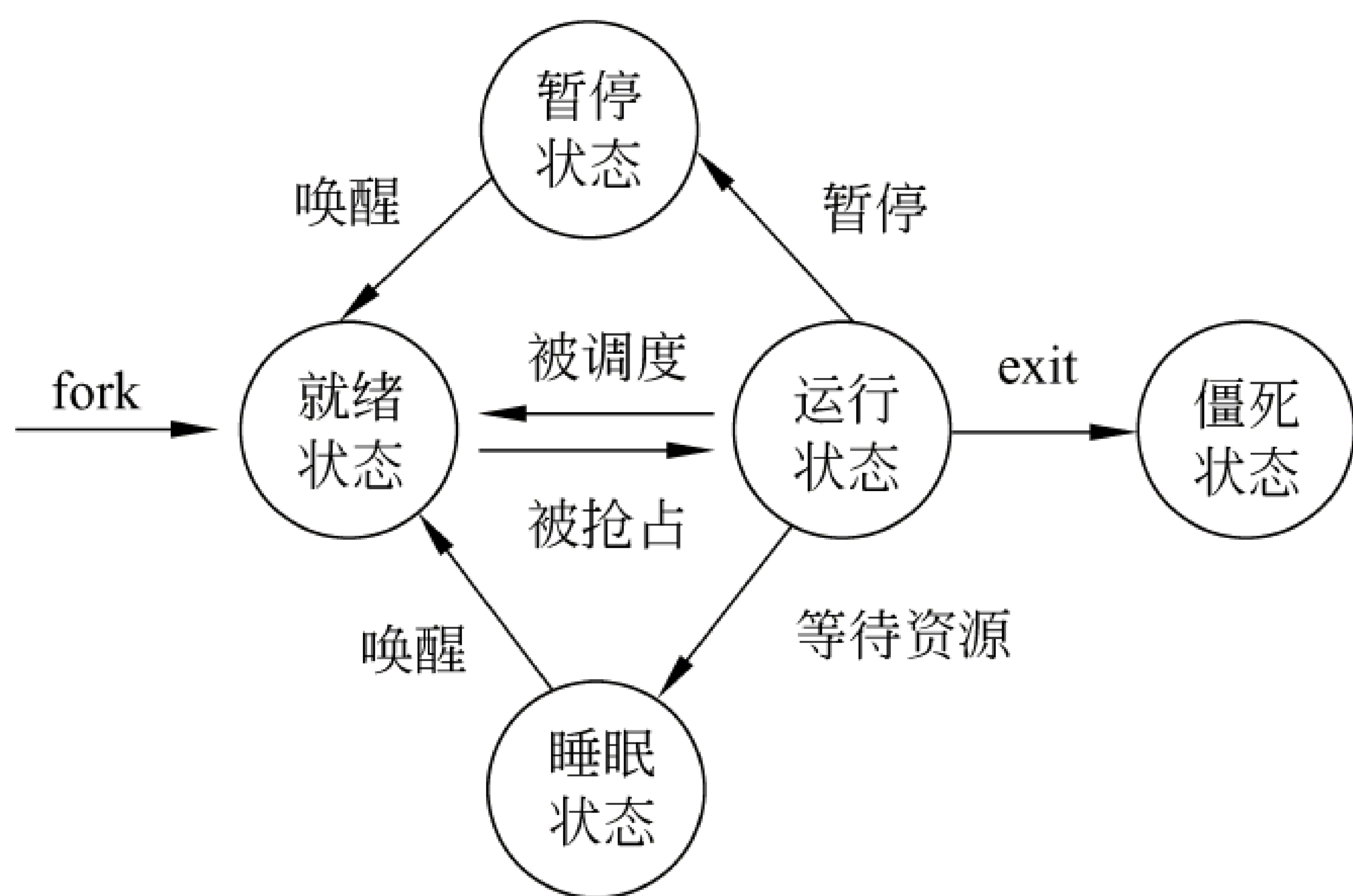



图 6.5　进程状态转换

为可中断的 TASK_INTERRUPTIBLE 状态和不可中断的 TASK_UNINTERRUPTIBLE 状态。在 Linux 2.6.26 内核版本中，TASK_ZOMBIE 状态分为 EXIT_ZOMBIE 和 EXIT_DEAD 两种，都保存在 exit_state 成员中。EXIT_ZOMBIE 是指进程已终止，正等待其父进程收集关于它的一些统计信息；EXIT_DEAD 是指进程的最终状态，即将进程从系统中删除时所处的状态。另外，Linux 2.6.26 以上的内核版本还有 TASK_DEAD 状态和 TASK_KILLABLE 状态。TASK_DEAD 状态是 EXIT_DEAD 状态的一种特殊情况，是在一个进程退出（调用 do_exit()）时所置的状态，是为了避免混乱而引入的；TASK_KILLABLE 状态是指可终止的进程睡眠状态，是可以响应致命信号的睡眠状态。为了在讨论进程状态转换的时候能够更清晰地描述整个状态的转换过程，这里将 Linux 的进程状态按照图 6.5 所示做了适当的划分和合并。

首先，当父进程执行系统调用 fork 时，核心为该进程分配 task_struct 结构并做必要的初始化工作。初始化完成后，该进程进入就绪状态。此时，由于该进程已经分得各种页表、堆栈、正文段和数据段所用的内存空间以及其他的系统资源，因此，该进程可以经调度选中后占有 CPU。

当进程进入就绪状态后，进程调度程序将会在适当时机选择该进程去执行，当进程被分配到 CPU 资源开始执行时，它处于运行状态。

处于运行状态的进程如果因为时间片耗尽或由于调度程序发现有更高优先级的进程而被抢占的时候，被剥夺 CPU 资源而回到就绪状态。关于进程调度和调度的时机等问题，在后面的章节中还会详细描述。

当进程处于运行状态时，用户程序中由于使用系统调用或由于输入输出数据等操作而等待某事件发生，例如等待输入输出完成时，进程会调用睡眠原语而置于睡眠状态。处于睡眠状态的进程直到事件发生后被唤醒原语唤醒而进入就绪状态。

还有一种情况，处于运行态的进程如果接受到某些软中断信号，例如 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU，系统会将进程置于暂停状态。其中 SIGSTOP 是由于调试程序调用 ptrace 系统调用调试目标程序而触发。用户在终端按 SUSP 键（在 PC 上通常是 Ctrl+Z 组合键）则会向前台进程发送 SIGTSTP 信号。后台进程如果访问控制终端的话，则会收到 SIGTTIN 或 SIGTTOU 信号。处于暂停状态的进程收到信号 SIGCONT 被唤醒而重新进入到就绪状态。

最后，在进程完成它所要求的任务之后，将使用系统调用 exit，从而使得进程进入僵死状态而释放资源。

### 6.2.5 小结

本节讨论了 Linux 中进程的虚拟地址空间，并介绍了进程的上下文和进程的状态及转换。进程的虚拟地址空间是进程管理和内存管理的基础，Linux 使用 AVL 平衡二叉树管理虚拟区域，使得对虚拟区域的操作更加高效。

进程上下文是进程的静态描述。进程描述符 task_struct 结构是系统感知进程存在的唯一实体。task_struct 结构包含了进程调度和运行所需要的大量的数据结构，并且常驻内存。进程上下文中的正文段和数据段是进程完成所要求任务的关键部分，通用寄存器、程序计数器以及处理机状态寄存器和页表等都是为了正文段的顺利执行而用来存放中间结果，或传递参数，或存放下一条指令的虚拟地址，或区别控制 CPU 的访问模式和完成地址变换等功能的。进程上下文各部分构成十分巧妙，缺一不可。

另外，进程可以在用户级对某些状态的转换加以控制。例如，用户可以创建进程，可以利用系统调用从用户态进入系统态。但是，有些状态转换是由系统控制的。例如，被创建进程是转换到运行态还是就绪态取决于调度过程。有些状态转换则要依靠外部事件和系统的共同控制。例如，一个处于睡眠态的进程何时能转换成就绪态则依赖于事件的发生和唤醒原语。

总之，进程的上下文和状态转换刻画了 Linux 进程的静、动两种特性，它们是进程的完整反映。

## 6.3 Linux 进程控制

### 6.3.1 Linux 启动及进程树的形成

在计算机启动后，首先得到处理的是 BIOS 或 EFI 等系统固件(firmware)，系统固件从磁盘的引导扇区加载引导加载程序，例如 GRUB 或 ELILO，引导加载程序负责将 Linux 系统的核心装入内存，并转到 Linux 核心所在的地址，开始 Linux 核心的初始化工作。第一步执行的是一些和硬件体系结构有关的代码，然后系统开始执行函数 start_kernel()。start_kernel()首先初始化系统内部数据结构，例如构造空闲缓冲区、初始化区结构和页表项等。然后，系统建立进程 0。进程 0 将根文件系统安装到根“/”下，创建新进程——进程 1，随后进程 0 的作用就转变为 idle 进程，只有在系统中没有任何进程可以被执行的时候才会得到调度。进程 1 在系统中被称为 init 进程。它是一个既可在核心态，又可在用户态运行的进程，它的正文代码由调用系统调用 exec 执行程序/sbin/init 的代码组成，init 进程负责初始化所有新的用户进程。

init 进程调用 exec 执行/sbin/init 程序后，为每个终端生成一个子进程，然后等待用户在终端上注册。

用户在终端上输入命令，每个命令都对应一个可执行文件。Shell 命令解释程序解释此命令，找到相应的可执行命令文件，用系统调用 fork 创建子进程，并由此子进程调用系统调

用 exec 执行此命令文件;父进程也就是 Shell 进程则处于等待状态,子进程执行文件结束后调用系统调用 exit 自我终止,唤醒父进程做善后处理并转而显示提示符,准备接收下一条命令执行。

进程启动及进程树的形成过程如图 6.6 所示。

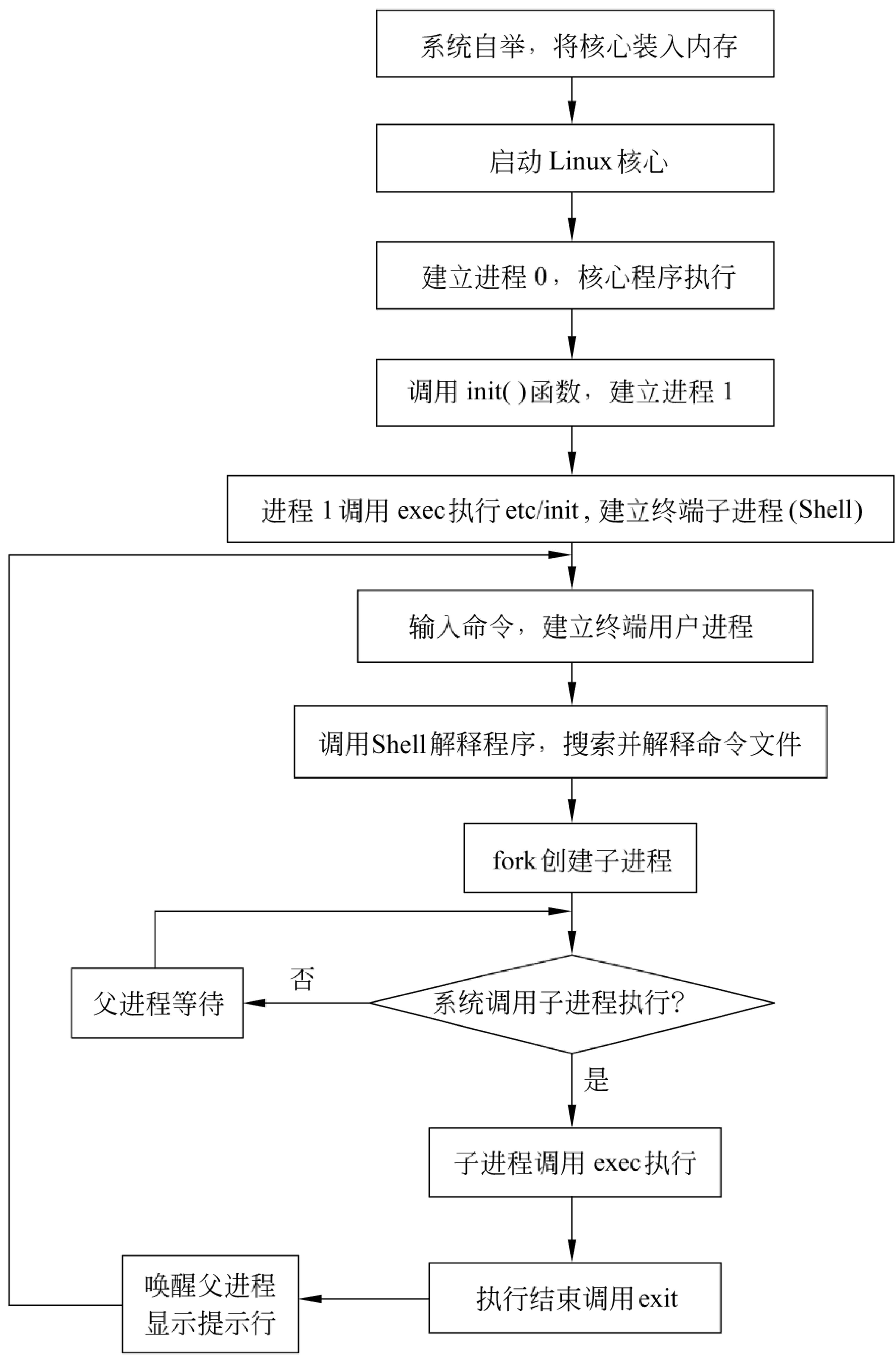



图 6.6　进程树的形成

## 6.3.2　进程控制

本节主要讨论用户进程的创建、执行和自我终止问题,与此相对应,Linux 系统提供了相应的系统调用 fork、exec 和 exit,以便在用户级上实现上述功能。

**1. 进程的创建**

fork 的功能是创建一个子进程。调用 fork 的进程称为父进程。

系统调用 fork 的语法格式是

```
pid=fork();
```

从系统调用 fork 返回时，父进程和子进程除了返回值 pid 与 task_struct 结构中某些特性参数不同之外，其他完全相同。CPU 在父进程中时，pid 值为所创建子进程的进程号；若在子进程中时，pid 的值为零。

有关 fork 系统调用的举例，已经在第 3 章中作了介绍，这里不再重复。为了便于理解 Linux 系统进程的并发性，下面介绍 fork 的功能与实现过程。

系统调用 fork 通过执行核心程序 fork 过程完成的功能如下：

（1）为子进程分配一个进程描述符 task_struct 结构，将父进程的进程描述符的内容复制到新创建的结构中，并重新设置那些与父进程不同的数据成员。

（2）为子进程分配一个唯一的进程标识号 pid。

（3）将父进程的地址空间的逻辑副本复制到子进程。这里的逻辑副本指的是写时复制(copy on write)机制。因为大多数情况下，子进程在创建以后会调用 exec 系统调用执行一个新程序，从而丢弃原有的地址空间，这样原来做的地址空间复制是一个极大的浪费。Linux 通过写时复制机制使子进程暂时共享父进程的有关数据结构，只有在子进程做修改的时候才会为其单独复制出一个副本，这样就减少了不必要的复制工作。

（4）复制父进程相关联的有关文件系统的数据结构和用户文件描述符表。这样子进程就继承了父进程的文件系统相关的信息。

（5）复制软中断信号有关的数据结构。

（6）设置子进程的状态为 TASK_RUNNING，把它加入到就绪队列，并启动调度程序。

（7）对父进程返回子进程的进程标识号，对子进程返回零。

下面介绍 fork 算法。其算法流程图如图 6.7 所示。

在图 6.7 中，系统首先为子进程分配一个进程描述符 task_struct 结构，将父进程的进程描述符的内容复制到新创建的结构中，并做必要的修改。

系统对用户同时拥有的进程个数有一定的限制，以免妨碍其他用户创建进程，所以在分配了子进程的进程描述符以后，需要对资源限额做一个检查，如果超出了进程资源限额的话就失败并返回。

下面，系统会为子进程分配一个新的进程号，新创建进程所分配的标识号按比最近一次分配的进程标识号大 1 的顺序递增。当标识号达到所规定的最大值之后则从 0 开始重新循环。

紧接着，系统复制父进程的文件系统指针和文件描述符表到子进程，使得子进程自动共享父进程所打开的文件，以及确定子进程在文件系统中所处的目录位置(有关文件系统，将在后面的章节中讲述)，同时复制父进程的软中断信号的处理函数指针。

然后，系统设置子进程的各种调度参数，并设置父进程的 need_resched 标志。

到这里子进程基本被初始化完毕，系统会修改相关进程的进程描述符的家族指针成员，以反映新的进程家族关系。

最后通过函数 wake_up_process()将子进程设置为运行状态，加入就绪队列，以便新的进程获得运行的机会。

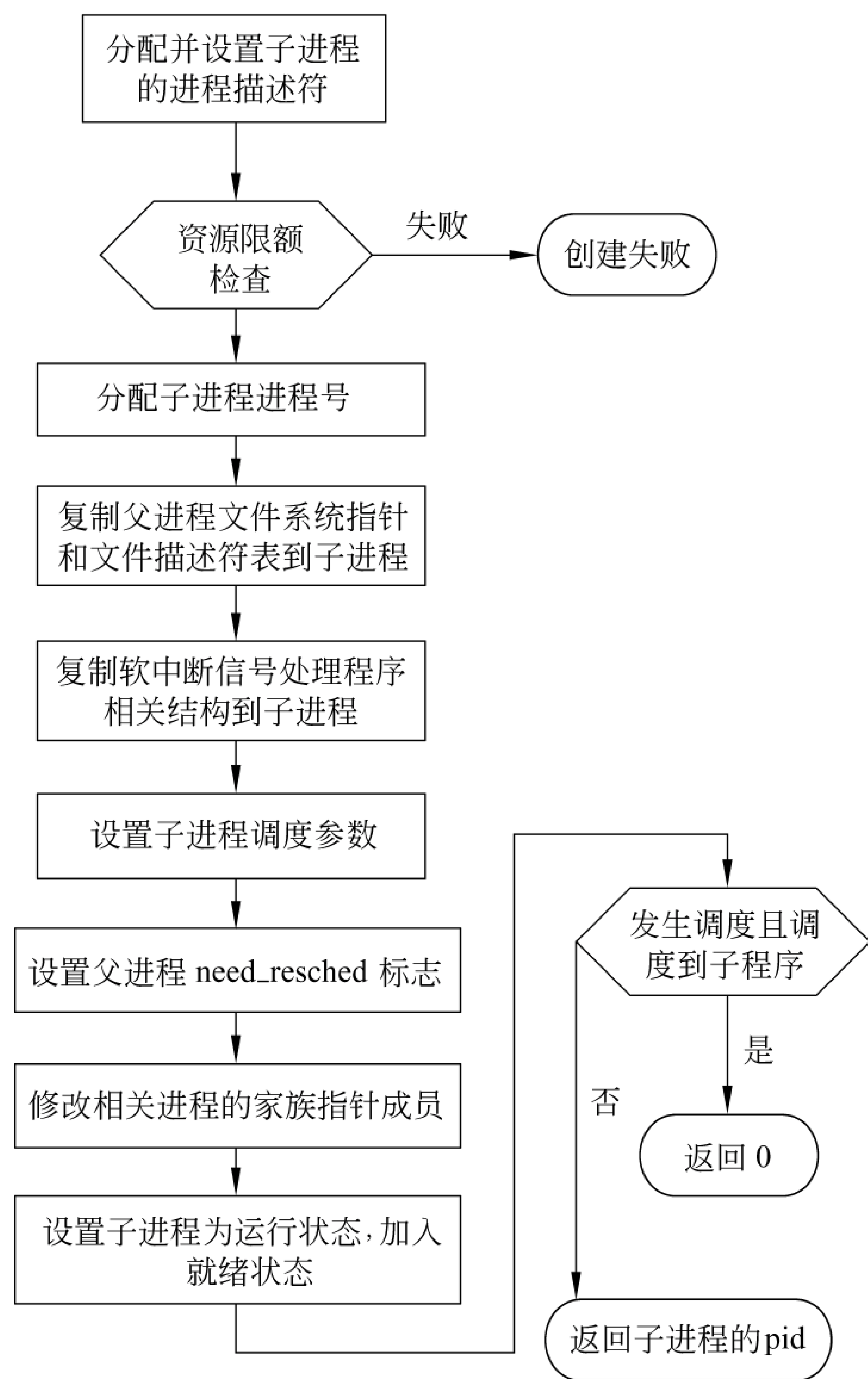



图 6.7　fork 算法流程图

**2. 执行一个文件的调用**

当父进程使用 fork 创建了子进程之后，子进程继承了父进程的正文段和数据段，从而限制了子进程可以执行的程序规模。那么，子进程用什么办法来执行那些不属于父进程的正文段和数据段呢？这就是利用系统调用 exec。系统调用 exec 引出另一个程序，它用一个可执行文件的副本覆盖调用进程的正文段和数据段，并以调用进程提供的参数转去执行这个新的正文段程序。

系统调用 exec 包含 6 种不同的调用格式，但它们都完成同一工作，即把文件系统中的可执行文件调入并覆盖调用进程的正文段和数据段之后执行。有关 exec 的各种系统调用的区别主要在参数处理方法上。这些系统调用使用不同的输入参数、环境变量和路径变量。这里，系统调用 execvp()和 execlp()在程序中经常用到，其调用格式是

```
execvp(filename, argp);
execlp(filename, arg0, arg1, …, (cbar * )0);
```

其中，filename 是要执行的文件名指针，argp 是输入参数序列的指针，而 0 则是参数序列的结束标志。

**例**：用 execlp 调用实现一个 Shell 的基本处理过程。

利用 fork 和 exec 可实现一个 Shell 的基本功能。用户输入命令后，按以下步骤执行用户命令（见图 6.8）。

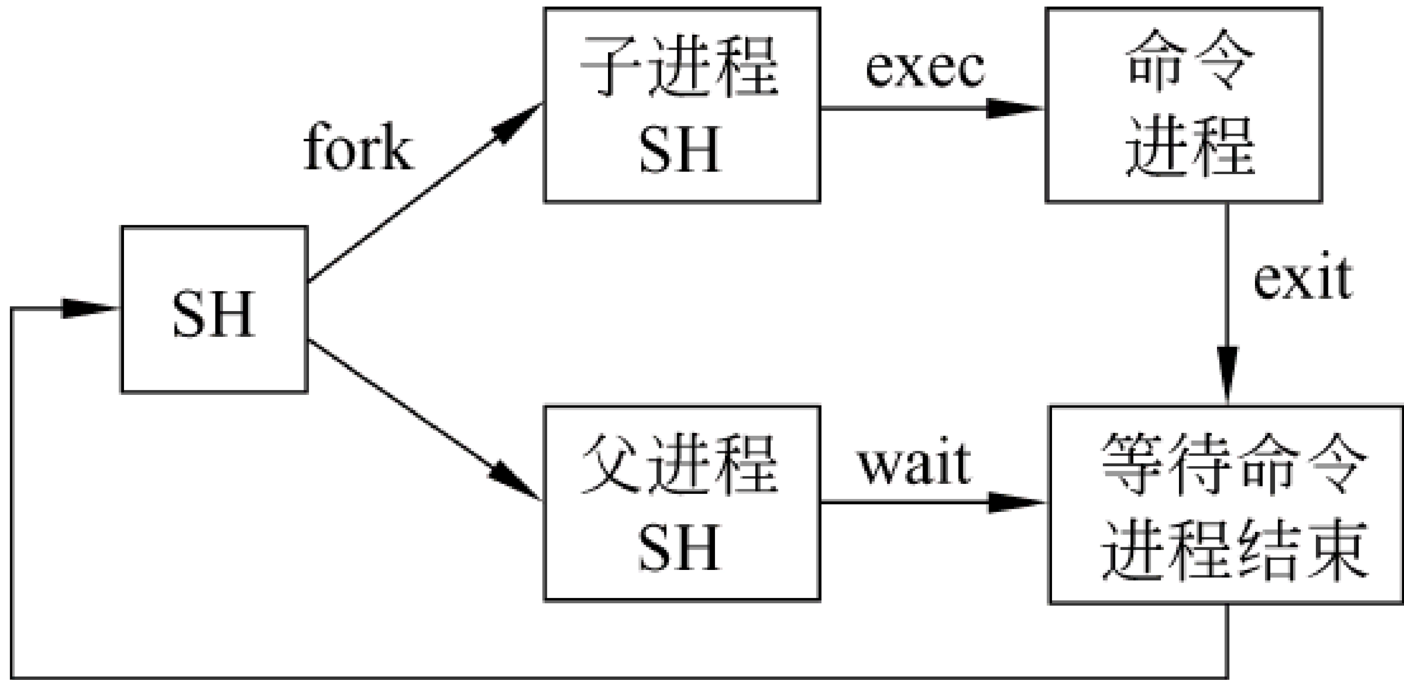



图 6.8　Shell 执行过程

（1）利用 fork，创建子进程。

（2）利用 exec，启动命令程序。

（3）利用 wait，父进程和子进程同步。

用 C 语言实现的程序代码如图 6.9 所示。

```
#include <stdio.h>
main()
{
    char  command[32];
    char  * prompt="$ ";
    while(printf ("%s", prompt), gets (command)!=NULL)
    {
        if (fork()==0)
            execlp (command, command, (char * )0);
        else
            wait(0);
    }
}
```

图 6.9　Shell 程序的实现代码

图 6.9 实现的 Shell 中不包含路径检索功能（即执行有关命令时，必须把从根开始至命令名为止的路径名全部输入），也没有包括参数处理功能。此例只是为了说明 exec 调用的应用以及 fork 与 exec 的关系。

**3. 进程的终止**

系统调用 exit(rv)自我终止当前进程，使其进入 ZOMBIE（僵死）状态，等待父进程进行善后处理。

exit 调用将导致释放除 task_struct 结构之外的所有资源，并清除进程上下文。父进程在收到子进程的信息 rv 和有关子进程的时间信息之后，将释放子进程的 task_struct 结构并将有关的时间信息加到自己的 task_struct 结构的有关项中去。

## 6.4　Linux 进程调度

Linux 系统的进程调度由核心的调度过程 schedule()实现。Linux 系统中没有三级调度中的高级调度，也没有中级调度。下面介绍进程的调度策略与实现。

**1. 调度原理**

Linux 系统的进程调度对实时进程和普通进程采用不同的调度算法。对于普通进程采用的是基于时间片的动态优先数调度法。即系统给进程分配一个时间片，当时间片结束时，动态计算该进程的优先级，若优先级高于当前内存就绪态进程时，系统设置调度标识，在核心态转换到用户态前由 schedule()过程调度优先级高的进程执行，并把被抢先的进程保存

到就绪队列中。Linux 的进程调度按时间片计算优先级，并按优先级的高低来调度进程抢占处理机。因此 Linux 系统的进程调度是基于时间片加优先级的。

进程调度涉及的主要问题如下：

（1）调度的时机。

（2）调度标志设置。

（3）调度策略与优先数的计算。

（4）调度的实现。

下面分别说明这几个问题。

**2. 调度的时机**

在 Linux 系统中，为了减少操作系统设计的复杂性和提高系统执行效率，只在核心的几个预定的位置进行调度。一种情况是，当处理机从核心态向用户态转换之前的瞬间，核心检查当前进程的调度标志 need_resched，如果该标志为 1，则运行调度过程，检查各就绪进程的优先级进行调度。从核心态向用户态转换的机会很多，例如从中断处理、陷阱处理（将在后面章节中讲述）和系统调用等返回。

另一种可以切换处理机的时机是当进程状态发生变化时，直接调用调度过程进行调度，例如，当进程因申请系统资源而未得到满足，从而调用 sleep() 放弃处理机；或者是进程为了与其他进程保持同步而调用 wait() 放弃了处理机；或由于执行了 exit 调用，终止了当前进程。这几种情况下，由于发生调度的时机是可以预见的，因此，在这些程序执行结束之前都主动调用调度过程调度其他优先级高的进程执行。

综上所述，Linux 中发生进程调度的时机实质上有两个：一个是进程自动放弃处理机时主动转入调度过程，另一个则是在由核心态转入用户态时，系统设置了高优先级就绪进程的强迫调度标识 need_resched 时发生调度。这两种情况下，调度过程都是指 schedule() 过程，后面会详细介绍这个过程。

在 Linux 2.6 核心版系统中，为支持嵌入式系统的开发和实时系统需求，除核心的上述两种调度时机之外，还在内核中增加了 3 种调度时机，允许调度程序中止当前进程而调用更高优先级的进程，这 3 种情况分别是：从中断或系统调用返回到用户态；某个进程允许被抢占 CPU；当进程主动进入休眠状态时。即 Linux 2.6 内核版本中进程调度在一定程度上是可抢占的，但也不是所有的内核代码段都可以被强占。

**3. 调度标识的设置**

UNIX System V 中有 3 个关于调度和交换用的调度标识，它们是 runrun、runin 和 runout。runrun 标识是要求处理机调度程序进行调度的标识，后两个标识和交换有关。Linux 只使用一个调度标识 need_resched，该标识保存在进程的进程描述符中。

在下面 3 种情况下 need_resched 标识会被设置为 1：

（1）当处于运行态的进程的时间片耗尽时，时钟中断处理程序会设置该进程的 need_resched 标识；

（2）当进程被唤醒，而它的优先级比正在运行的当前进程的优先级高的情况下，当前进程的 need_resched 标识会被置位；

（3）当一个进程通过系统调用改变调度政策和 nice 值等时被置位。

**4. 调度策略与优先数的计算**

Linux 把进程分为普通进程和实时进程，实时进程的调度优先级比普通进程的要高，

Linux 总是优先调度实时进程，以便满足实时进程对响应时间的要求。相应地，Linux 使用 3 种调度策略，动态优先数调度 SCHED_OTHER、先来先服务调度 SCHED_FIFO 和轮转法调度 SCHED_RR。其中动态优先数调度策略用于普通进程，后两种调度策略用于实时进程。进程可以通过 sched_setscheduler()系统调用选择适合自己的调度策略。如果选择了两种实时调度中的任何一种，该进程就转变为一个实时进程。进程的调度策略保存在进程描述符中，并且被子进程所继承，所以实时进程的子进程仍然是一个实时进程。

1）动态优先数调度策略

Linux 核心通过 goodness()函数计算进程的优先级数。动态优先数调度策略从内存就绪队列中选取优先数最大的进程投入执行。

goodness()采用下式计算各进程的优先数：

$$weight = counter + priority - nice$$

其中，counter 是进程可用的时间片的动态优先级，进程创建时父进程的 counter 值被平分为两部分，一半被保留给父进程，另一半则由子进程获得。由此可见 fork 出子进程后，父子进程拥有的时间片并不会增加，这样避免了用户通过 fork 子进程耗尽系统的 CPU 资源。在时间中断中，当前进程的 counter 被减少，这样就逐渐地降低了正在运行中的进程的优先级，使处于就绪队列的低优先级的进程得到运行机会。

如果所有运行队列中的进程的时间片都用完了，则调度程序会重新计算所有进程（不仅仅是运行状态的进程）可用的时间片的优先级，计算公式是

$$counter = counter/2 + nice$$

这样处于等待或睡眠状态的进程会周期性地得到提升优先级别的机会。

priority 值是一个常数，固定为 20。

nice 是系统允许用户设置的一个进程优先数偏移值。Linux 中，nice 值默认为 0，但是进程可以通过系统调用 nice()设置成－20～19 间的一个数。Linux 规定只有超级用户可以通过 nice 系统调用提高进程的运行优先级，而普通用户只能降低进程的运行优先级。

2）先来先服务调度策略

先来先服务调度策略最早进入就绪队列的进程，此策略中，被调度的进程一直运行，直到具有更高优先级的进程进入就绪队列或当前进程结束或阻塞。如果此进程被抢占，它继续处于其优先级队列的首部，如果阻塞的话，当它再次成为就绪进程，将被添加到它所处的优先级队列的尾部。

3）轮转法调度策略

轮转法调度基本和先来先服务调度相同，不同的地方是进程只执行一个时间片，时间片一到，该进程就被加入到它所处的优先级队列的尾部。

4）0/1 调度策略

为支持嵌入式系统开发和多处理器并行处理，Linux 系统中还提供了一种新的调度策略——0/1 调度策略。在系统中设置了两个队列，分别为活动队列和过期队列，任务就绪时被放入活动队列中，调度程序每隔一定时间从活动队列取任务，任务运行时分配一个时间片，当时间片结束时，该任务放弃处理机并根据其优先级转入过期队列中。当活动队列中的任务全部调度结束后，两个队列的指针互换，过期队列成为当前活动队列，调度程序继续调度当前队列的任务。

**5. 调度的实现**

进程调度是由 schedule()过程实现的。关于调用 schedule()过程的时机，已在前面做

了介绍。这里主要讲述 schedule 的执行过程。调度过程分为两个阶段。

第一个阶段是进行进程选择。首先遍历进程,如果进程采用的是 SCHED_RR 策略,且时间片已用完,则重新为其分配新的时间片并移动到队列的末尾。如果进程的状态是 TASK_INTERRUPTIBLE 且收到了软中断信号,则将其恢复为就绪状态。如果所有的就绪进程的时间片都用完了,则重新计算所有进程的时间片。最重要的工作则是按照上述调度策略从所有进程中找到具有优先数最高的进程,当找到合适的进程后第一阶段就结束了。

第二个阶段是进程的切换过程,主要完成进程的上下文切换,其中用户进程的地址空间的切换是由 switch_mm()过程完成的,而进程的堆栈切换过程则由 switch_to()实现。switch_to()过程和硬件的体系结构有关,在 80x86 中这部分是一个很短的汇编语言代码,在切换了堆栈以后,进程从上一次被中断的位置继续执行。

## 6.5 Linux 进程通信

Linux 中的进程通信分为 3 个部分:低级通信、管道通信和进程间通信(Inter-Process Communication,IPC)。Linux 同时支持计算机间通信(网络通信)用的 TCP/IP 协议并提供了相应的系统调用接口。有关 TCP/IP 协议及其有关系统调用已超出了本书的范围,这里不作介绍。另外,关于管道通信已在第 3 章中作了介绍,这里不再重复。

### 6.5.1 Linux 的低级通信

Linux 的低级通信主要用来传递进程间的控制信号,主要是文件锁和软中断信号机制。软中断信号的目的是通知对方发生了异步事件。Linux 中有 31 个软中断信号和 32 个实时软中断信号,软中断信号的作用见表 6.1。实时软中断信号的编号为 32～63。它们没有预先定义的含义,和普通软中断信号的区别在于实时信号可以排队而不会发生丢失的现象。

**表 6.1 Linux 软中断信号**

| 软中断号 | 符号名 | 功　　能 | 软中断号 | 符号名 | 功　　能 |
|---|---|---|---|---|---|
| 1 | SIGHUP | 用户终端连接结束 | 17 | SIGCHLD | 子进程消亡 |
| 2 | SIGINT | 键盘按 DELETE 键 | 18 | SIGCONT | 继续进程的执行 |
| 3 | SIGQUIT | 键盘按 QUIT 键 | 19 | SIGSTOP | 停止进程的执行 |
| 4 | SIGILL | 非法指令 | 20 | SIGTSTP | 键盘按 SUSP 键 |
| 5 | SIGTRAP | 断点或跟踪指令 | 21 | SIGTTIN | 后台进程读控制终端 |
| 6 | SIGABRT | 程序 ABORT | 22 | SIGTTOU | 后台进程写控制终端 |
| 7 | SIGBUS | 非法地址 | 23 | SIGURG | socket 收到紧急数据 |
| 8 | SIGFPE | 浮点溢出 | 24 | SIGXCPU | 超过 CPU 资源限制 |
| 9 | SIGKILL | 要求终止该进程 | 25 | SIGXFSZ | 超过文件资源限制 |
| 10 | SIGUSR1 | 用户定义 | 26 | SIGVTALRM | 虚拟时钟定时信号 |
| 11 | SIGSEGV | 段违例 | 27 | SIGPROF | 虚拟时钟定时信号 2 |
| 12 | SIGUSR2 | 用户定义 | 28 | SIGWINCH | 窗口大小改变 |
| 13 | SIGPIPE | PIPE 只有写者,无读者 | 29 | SIGIO | IO 就绪 |
| 14 | SIGALRM | 时钟定时信号 | 30 | SIGPWR | 电源失效 |
| 15 | SIGTERM | 软件终止信号 | 31 | SIGSYS | 系统调用错 |
| 16 | SIGSTKFLT | 协处理器的堆栈异常 | | | |

软中断是对硬件中断的一种模拟，发送软中断就是向接收进程的 task_struct 结构中的相应项发送表 6.1 中的一个信号。接收进程在收到软中断信号后，将按照事先的规定去执行一个软中断处理程序。但是，软中断处理程序不像硬中断处理程序那样收到中断信号后立即被启动，它必须等到接收进程执行时才能生效。另外一个进程也可以向自己发送软中断信号，以便在某些意外的情况下，进程能转入规定好的处理程序。例如，大部分陷阱都是由当前进程向自己发送一个软中断信号而立即转入相应处理的。

为了给用户进程也提供相应的同步、互斥以及软中断通信功能，Linux 系统提供了几种相应的系统调用或库函数。其中文件锁库函数 lockf()可以用于互斥，其格式是

```
lockf(fd, function, size)
```

其中，fd 是被锁定文件标识，文件标识必须使用只写权限(O_WRONLY)或读写权限(O_RDWR)打开；function 是控制值，F_LOCK 表示锁定一个文件的某个区域，F_UNLOCK 表示不再锁定，F_TLOCK 用来测试和锁定一个程序段，F_TEST 则用来测试待锁定的程序段是否已被其他进程锁定。size 表示要锁定或解锁的连续字节数，如果 size 等于零，则表示锁定从调用 lockf 后到文件结尾的区域。

lockf 在 Linux 中是通过 fcntl 系统调用实现的，用户进程也可以直接用 fcntl 系统调用使用文件锁。

用于同步的系统调用是 wait()或 sleep($n$)。其中，wait()用于父子进程之间的同步，而 sleep 则使得当前进程睡眠 $n$ 秒后自动唤醒自己。

系统调用 kill(pid, sig)和 signal(sig, func)被用来传递和接收软中断信号。一个用户进程可调用 kill(pid, sig)向另一个标识号为 pid 的用户进程发送软中断信号 sig。根据表 6.1 可知，用户可以定义的软中断号是 10 或 12。另外，标识号为 pid 的进程通过 signal(sig,func)捕捉到信号 sig 之后，执行预先约定的动作 func，从而达到这两个进程通信的目的。一个经常用到的例子是 signal(SIGINT, SIG_IGN)，表示当前进程不做任何指定的工作而忽略键盘中断信号的影响。

### 6.5.2 进程间通信

UNIX System Ⅴ 版本设计了一套进程间通信(IPC)的机制，后来被称为 System Ⅴ IPC。它解决了 UNIX 早期版本在进程间通信方面的弱点。在 System Ⅴ IPC 机制被开发出来之前，通信能力一直是 UNIX 系统的一个弱点，因为只能利用 pipe 来传递大量数据。而 pipe 又存在着只有调用 pipe 的进程的子孙后代才能使用它进行通信的缺点。虽然有名管道能使非同族进程之间相互通信，但它们不能复用一个有名管道以便为多对通信进程提供私用通道。也就是说有名管道不能识别其通信伙伴，也不能有选择地接收信息。System Ⅴ IPC 机制解决了这些弱点，Linux 则完整地继承了 System Ⅴ IPC。

System Ⅴ IPC 有 3 个组成部分：

(1) 消息(message)用于进程之间传递分类的格式化数据。

(2) 共享存储器(shared memory)方式可使得不同进程通过共享彼此的虚拟空间而达到互相对共享区操作和数据通信的目的。

(3) 信号量(semaphore)机制用于通信进程之间的同步控制。信号量通常与共享存储

器方式一起使用。

由于上述 3 种方式是作为一个整体实现的，因此，它们具有下述共同性质：

（1）每种机制都用两种基本数据结构来描述：

① 索引表。其中一个表项由关键字、访问控制结构及操作状态信息组成。每个索引表项描述一个通信实例或通信实例的集合。

② 实例表。一个实例表项描述一个通信实例的有关特征。例如，消息机制中消息队列表相当于索引表，而消息头表则相当于实例表，如图 6.10 所示。

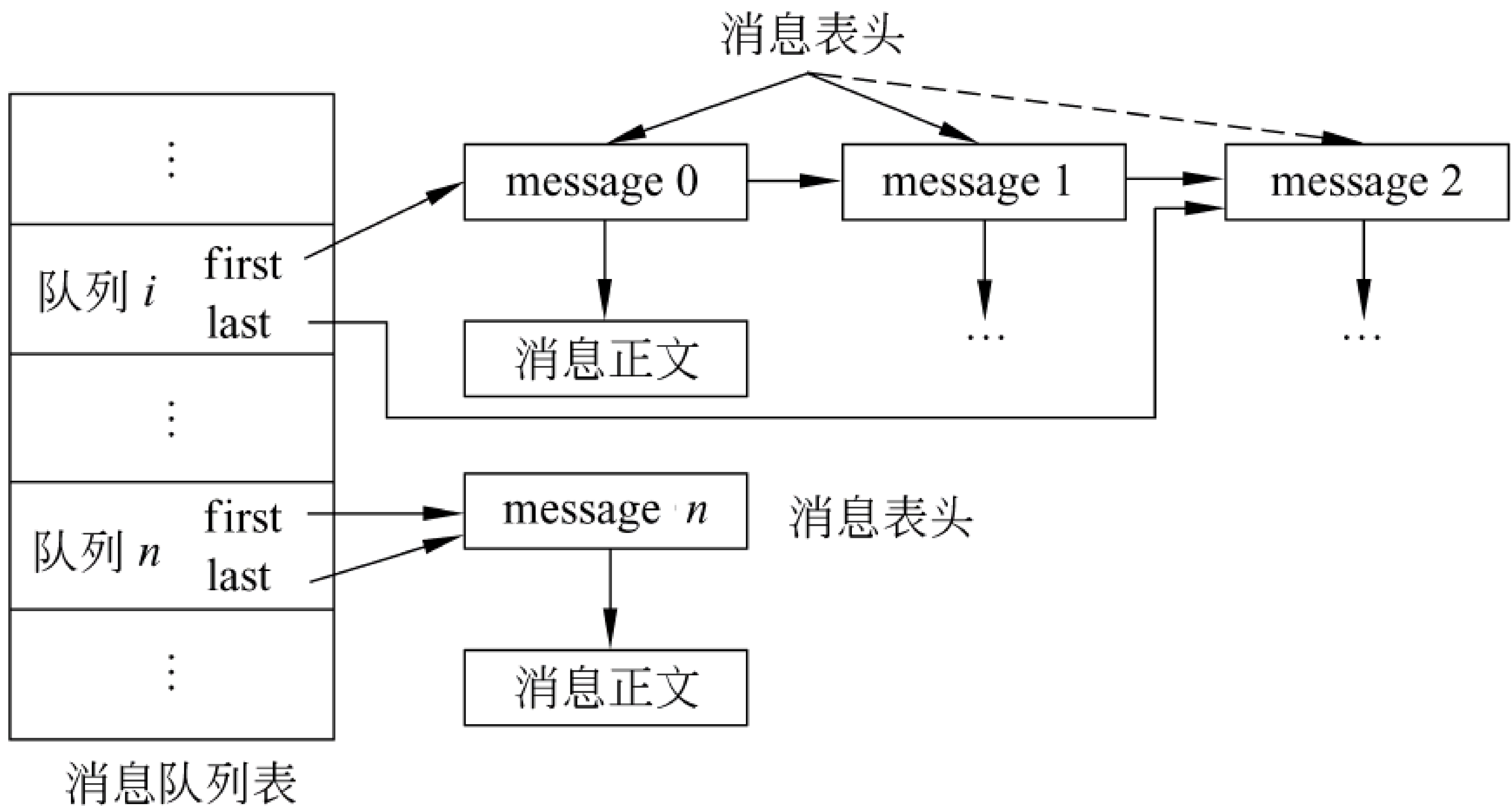



图 6.10　索引表与实例表的关系

（2）索引表项中的关键字是一个大于零的整数，它由用户选择名字。

（3）索引表的访问控制结构中含有创建该表项进程的用户 id 和用户组 id，由后述 control 类系统调用，可为用户和同组用户设置读-写-执行许可权。从而起到通信保护的作用。

（4）每种通信机制的 control 类系统调用可用来查询索引表项中的状态，以及置状态信息或从系统中删除表项。

（5）除了 control 类系统调用之外，每种通信机制还含有一个 get 类系统调用，以创建一个新的索引表项或者用于获得已建立的索引表项的描述字。

（6）每一种索引表项都使用下列公式计算索引表项的描述字。

描述字＝SEQ_MULTIPLIER×分配序号＋索引表项下标

其中 SEQ_MULTIPLIER 是索引表长度的上限，目前为 32768。例如，表项 1 的描述字可以是 32769、65537、98305 等。这样做的好处是，当进程释放了一个旧的索引表项，且该索引表项又分配给另外的进程时，因为分配序号的增加将使得描述字改变，从而原来的进程不可能再次访问该表项。由此，可以起到通信保护作用。

其他系统调用访问索引表项时的索引值为(描述字)mod(索引表长度)。

下面，简单地介绍 3 种通信机制的系统调用。

**1. 消息机制**

消息机制提供 4 个系统调用：

```
int msgget(key_t key, int msgflg);
int msgctl(int msgqid, int cmd, struct msqid_ds * buf);
int msgsnd(int msgqid, struct msgbuf * msgp, size_t msgsz, int msgflg);
```

```
ssize_t msgrcv(int msgqid, struct msgbuf * msgp, size_t msgsz, long msgtyp, int msgflg);
```

这些系统调用所需要的数据结构和表格都放在头文件<sys/types.h>、<sys/ipc.h>和<sys/msg.h>中。因此，在使用各种通信机制的系统调用之前，必须include这3个头文件。

系统调用msgget返回一个消息描述字msgqid，msgqid指定一个消息队列以便其他3个系统调用使用。key和msgflg具有获取的语义。key可以等于关键字IPC_PRIVATE，以保证返回一个未用的空表项，key还可以被设置成一个还不存在表项描述字的表项号。这时，只要msgflg&IPC_CREAT为真，则系统会生成一个新的表项并返回描述字。例如：

```
msgqid=msgget (MSGKEY, 0777)
```

在MSGKEY所对应的消息队列表项不存在时，将创建该表项；在MSGKEY所对应的表项存在时，msgget返回该表项的描述字。

系统调用msgctl用来设置和返回与msgqid相关联的参数选择项，以及用来删除消息描述符的选择项。cmd的取值范围为{IPC_STAT，IPC_SET，IPC_RMID}。其中，IPC_SET表示将指针为buf中的用户id等读入与msgqid相关联的消息队列表项中；IPC_STAT表示将与msgqid相关的消息队列表项中所有当前值读入buf所指的用户结构中；而IPC_RMID则表示msgctl调用删除msgqid所对应的消息队列表项。buf是用户空间中用于设置或读取消息队列状态的索引结构指针。

系统调用msgsnd和msgrcv则分别表示发送和接收一个消息。msgsnd(msgqid，msgp，msgsz，msgflg)中的msgqid是msgget返回的消息队列描述符；msgp则是用户消息缓冲区指针；msgsz是消息正文的长度；而msgflg则是同步标识，规定msgsnd发送消息时，是发送完毕后返回还是不等发送完毕立即返回(此时msgflg&IPC_NOWAIT为真)。

系统调用msgrcv中比msgsnd多一个参数msgtyp，它规定接收消息的类型。msgtyp=0时，表示接收与msgqid相关联的消息队列上的第一个消息；msgtyp>0时，表示接收与msgqid相关联的消息队列上msgtyp类型的第一个消息；而msgtyp<0时，则表示接收小于或等于msgtyp绝对值的最低类型的第一个消息。另外，msgflg指示当与msgqid相关联的消息队列上无消息时系统应当怎么办。

例如，图6.11和图6.12分别给出了用C语言编写的，由顾客进程和服务者进程调用的程序例子。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY 75
struct msgform
{
    long mtype;
    char mtext [256];
};
```

图6.11　顾客进程的程序段

```
main ()
{
    struct msgform msg;
    int msgqid, pid, * pint;
    msgqid=msgget (MSGKEY, 0777);            /* 建立消息队列 */
    pid=getpid ();
    pint= (int * ) msg.mtext;
     * pint=pid;
    msg.mtype=1;                             /* 指定消息类型 */
    msgsnd(msgqid,&msg,sizeof(int),0);       /* 往 msgqid 发送消息 msg */
    msgrcv(msgqid,&msg,256,pid,0);           /* 接收来自服务进程的消息 */
    printf("client : receive from pid%d\n", * pint);
}
```

图 6.11（续）

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MSGKEY 75
struct msgform
{
    long mtype;
    char mtext [256];
} msg;
int msgqid;
main ()
{
    int i, pid, * pint;
    extern cleanup ();
    for (i=0; i<20; i++)                     /* 软中断处理 */
      signal (i, cleanup);
    msgqid=msgget (MSGKEY, 0777|IPC_CREAT);  /* 建立与顾客进程相同的消息队列 */
    for (;;)
    {
        msgrcv (msgqid, &msg, 256, 1, 0);    /* 接收来自顾客进程的消息 */
        pint= (int * ) msg.mtext;
        pid= * pint;
        printf("server: receive from pid %d\n", pid);
        msg.mtype=pid;
         * pint=getpid ();
        msgsnd (msgqid, &msg, sizeof(int), 0);   /* 发送应答消息 */
    }
}
cleanup ()
{
    msgctl (msgqid, IPC_RMID, 0);
    exit ();
}
```

图 6.12　服务者进程的程序段

图 6.11 是顾客进程。该进程向服务者进程发送一个含有进程号 pid 以及类型为 1 的消息,向服务者进程发出服务请求。然后,从服务者进程接收相应的回答或服务。

在图 6.11 中,MSGKEY 是用户自己定义的关键字,已在本书前面部分做过说明。而 msgform 则是用户自定义的发送消息正文和消息类型,这一消息被定义为 256B 长。紧接着,顾客进程首先使用系统调用 msgget 创建,或得到与关键字 MSGKEY 相关联的消息队列描述字 msgqid,并由库函数 getpid()得到该进程 ID。接下去是对消息正文做类型转换,以便计算消息长度并将进程 ID 复制到消息正文中。最后,顾客进程调用 msgsnd 把消息 msg 挂入以 msgqid 为描述字的消息队列,并从该队列接收服务进程发往该进程的第一个消息(用进程号 pid 作消息类型)。

图 6.12 是服务者进程。该进程首先检查是否捕捉到由 kill 发来的软中断信号。如果捕捉到时,则调用函数 cleanup 从系统中删除以 msgqid 为描述字的消息队列。如果它捕捉不到软中断信号或者接收的是不能捕捉的 SIGKILL(9)信号,则该消息队列继续保留在系统中,而且在该队列被删除以前试图以该关键字建立新消息队列的尝试都会失败。

然后,服务者进程使用系统调用 msgget,并在 msgget 中置位 IPC_CREAT 来建立一个消息队列结构。紧接着,服务者进程接收所有类型为 1 的,也就是从顾客进程来的请求消息。这是由系统调用 msgrcv 完成的。在接收到消息之后,服务者进程从消息中读出顾客进程的 ID,并将返回的消息类型置为顾客进程的 ID。然后,服务者进程把要发送的消息复制到消息正文域中,并使用 msgsnd 将消息挂入 msgqid 为描述字的消息队列。本例中,由服务者进程发送给顾客进程的消息也是服务者进程的 ID。

在消息机制中,消息被格式化为类型与数据对,且允许不同的进程根据不同的消息类型进行接收,这是使用管道通信所无法办到的。

**2. 共享存储区机制**

进程能够通过共享虚拟地址空间的若干部分,然后对存储在共享存储区中的数据进行读和写来直接地彼此通信。操纵共享存储区的系统调用类似于消息机制,共有 4 个系统调用:

```
int shmget(key_t key, size_t size, int shmflg);
void * shmat(int shmid, const void * shmaddr, int shmflg);
int shmdt(const void * shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds * buf);
```

shmget 建立新的共享区或返回一个已存在的共享存储区描述字。其中,key 是用户指定的共享区号,size 是共享存储区的长度,而 shmflg 则与 msgget 中的 msgflg 含义相同。

shmat 将物理共享区附接到进程虚拟地址空间。其中 shmid 是 shmget 返回的共享区描述字,而 shmaddr 是将共享区附接到其上的用户虚拟地址,当 shmaddr=0 时系统自动选择适当地址进行附接(默认)。shmflg 规定对此区是否是只读的,以及核心是否应对用户规定的地址作舍入操作。shmat 返回系统附接该共享区后的虚拟地址。

shmdt 进程从其虚拟地址空间断接一个共享存储区。其中,shmaddr 是 shmat 返回的虚拟地址。

shmctl 查询及设置一个共享存储区状态和有关参数。其中,shmid 是共享存储区的描述字,cmd 规定操作类型,而 buf 则是用户数据结构的地址,这个用户数据结构中含有该共

享存储区的状态信息。

用户可以使用上述 4 个系统调用为通信进程建立、附接以及断接共享存储区。共享存储区的好处在于为通信进程提供了直接通信的手段，使得通信进程可以直接访问彼此的某些虚拟空间。这既减少了数据流动所带来的硬软件开销（例如缓冲区及其管理等），又使得彼此的通信不仅仅局限于数据的发送与接收，而且可以互相操作彼此的某些虚拟存储区。

图 6.13 给出两进程共享存储区的示意图。

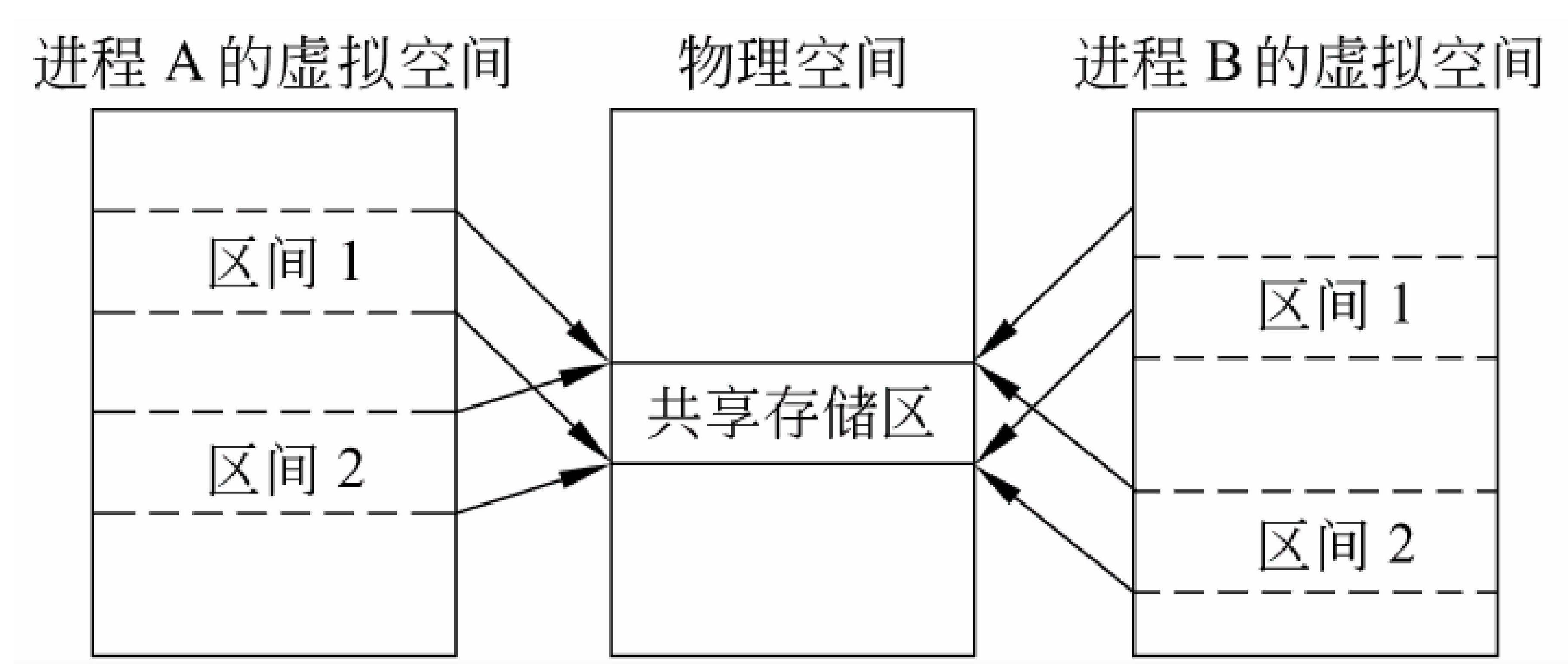



图 6.13　共享存储区示意图

一个共享存储区建立后可以被附接到一个进程的多个虚拟区间上，而一个进程的虚拟空间也可以附接多个共享存储区。

需要指出的是，共享存储区机制只为通信进程提供了访问共享存储区的操作条件，而对通信的同步控制则要依靠后述的信号量机制等才能完成。

图 6.14 和图 6.15 分别给出了将进程附接到共享存储区上，以及进程间共享存储区的 C 语言程序实例。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
#define K      1024
int shmid;
main ()
{
    int i, *pint;
    char *addr;
    extern char *shmat();
    extern cleanup();
    for (i= 0; i<20; i++)                          /*软中断处理*/
      signal (i, cleanup);
    shmid=shmget (SHMKEY, 16*K, 0777|IPC_CREAT);   /*建立 16KB 共享区 SHMKEY*/
    addr=shmat (shmid, 0, 0);                      /*共享区首地址*/
    printf ("addr 0x%x\n", addr);
    pint=(int*)addr;
    for (i=0; i<256; i++)
       *pint++=i;
    pint=(int*)addr;                 /*共享区第一个字中写入长度 256,以便接收进程读*/
    *pint=256;
```

图 6.14　共享存储区程序实例

```
    pause ();                                    /* 等待接收进程读 */
}
cleanup ()
{
    shmctl (shmid, IPC_RMID, 0);
    exit ();
}
```

图 6.14（续）

在图 6.14 中，该进程建立了 16KB 的共享存储区，并将存储区附接到了虚拟地址 addr 上。然后，从该存储区的起始单元开始，顺序写入 0～255 个自然数。如果该进程捕捉到一个软中断信号（SIGKILL 除外），则由系统调用 shmctl 删除该共享区。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY  75
#define K       1024
int shmid;
main ()
{
    int  i, *pint;
    char *addr;
    extern char *shmat ();
    shmid=shmget(SHMKEY, 8*K, 0777);      /* 取共享区 SHMKEY 的 id */
    addr=shmat (shmid, 0, 0);             /* 连接共享区 */
    pint=(int *)addr;
    while (*pint==0);                     /* 共享区的第一个字节为零时，等待 */
      for (i=0; i<256; *pint++)           /* 打印共享区中内容 */
        printf ("%d\n", *pint++);
}
```

图 6.15　共享存储区程序实例

在图 6.15 中另一个进程附接到与关键字 SHMKEY 相关联的存储区上。也就是与图 6.14 所述的同一个存储区上。为了表明每个进程可以附接一个共享存储区的不同总量，图 6.15 中只取该存储区的 8KB。该进程等待着直到图 6.14 中进程在共享存储区中的第一个字节写入一个非零值后读出该存储区；此时图 6.14 进程暂停以使图 6.15 进程执行读出打印操作。

**3. 信号量机制**

信号量机制是基于第 3 章所述的 P、V 原语原理的。UNIX System V 中一个信号量由以下几部分组成：

（1）信号量的值，是一个大于、小于或等于零的整数。

（2）最后一个操纵信号量的进程的进程 id。

（3）等待着信号量值增加的进程数。

（4）等待着信号量值等于零的进程数。

信号量机制提供下列系统调用对信号量进行创建、控制以及 P、V 操作是：

(1) 用于产生一个信号量数组以及得以存取它们的系统调用 semget(semkey, count, flag)。其中,semkey 和 flag 类似于建立消息和共享存储区时的这些参数。semkey 是用户指定的关键字,count 规定信号量数组的长度(如图 6.16 所示),flag 为操作标识。semget 用来创建信号量数组或查找已创建信号量数组的描述字。例如:

```
semid=semget(SEMKEY, 2, 0777|IPC_CREAT);
```

创建一个关键字为 SEMKEY 的含有两个元素的信号量数组。

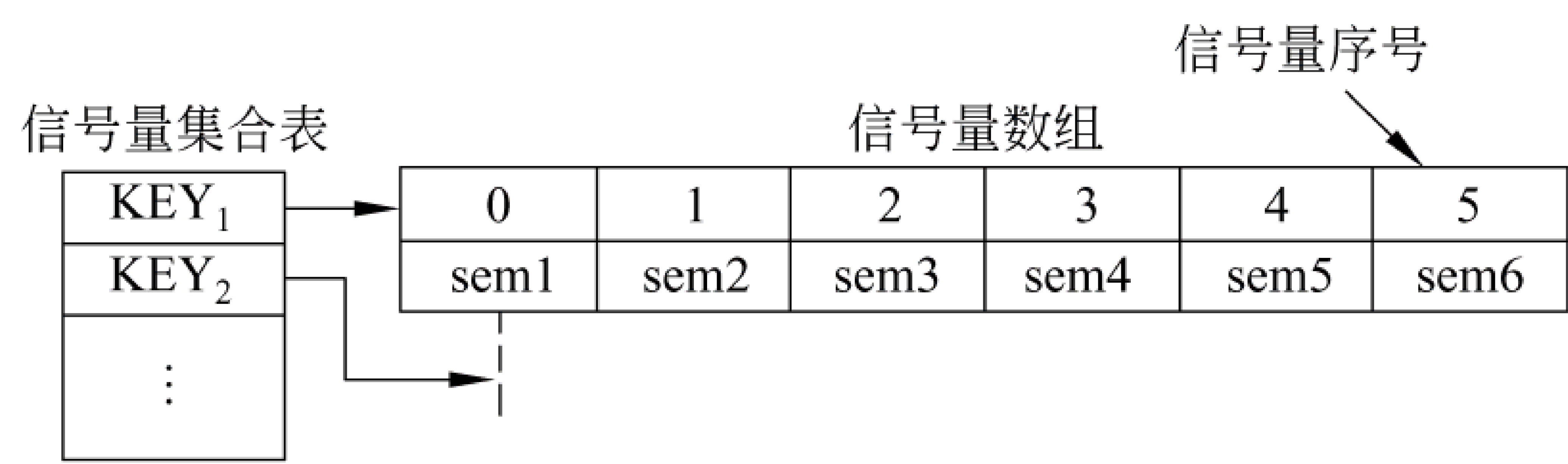



图 6.16　信号量数组

(2) 用于 P、V 操作的系统调用 semop(semid, oplist, count)。其中,semid 是 semget 返回的描述字,oplist 是用户提供的操作数组的指针,count 是该数组的大小。semop 返回在该组操作中最后被操作的信号量在操作完成前的值。

用户定义的操作数组中的每个元素包含 3 个内容,它们是信号量序号、操作内容(对信号量进行 P 操作或 V 操作的值)和标识。一个数组可同时包含对 $n(n>1)$ 个信号量的操作。

semop 根据操作数组所规定的操作内容改变信号量的值。如果操作内容为正数(V 操作),则将该信号量增加该操作内容的值,并唤醒所有等待此信号量值增加的进程。如果操作内容为零,则 semop 检查信号量的值,若为零,semop 执行对同一操作数组中其他信号量的操作;否则,废弃本次系统调用所完成的所有信号量操作之后,调用 sleep 使该进程进入睡眠状态。如果操作内容是负数且绝对值小于等于信号量值,则 semop 从信号量值中减去操作内容;否则,调用 sleep 让该进程睡眠在等待信号量值增加的事件上。例如:

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
}  Psembuf;
semid=semget (SEMKEY ,2 ,0777);
Psembuf.sem_num=first;
Psembuf.sem_op=-1;
Psembuf.sem_flg=SEM_UNDO;
semop (semid ,&Psembuf,1);
```

定义了一个对二元信号量数组中第一个信号量的 P 操作。其中 SEM_UNDO 是为了保证 P 操作的原子性而设置的标识。

(3) 对信号量进行控制操作的系统调用 semctl(semid, number, cmd, arg)。其中,semid 是 semget 返回的信号量的描述字,number 是对应于 semid 的信号量数组的序号,cmd 是控制操作命令,arg 是控制操作参数,是一个 union 结构。

```
union semun {
    int val
    struct semid_ds * buf
    unsigned short * array;
}arg
```

系统根据 cmd 的值解释 arg,并完成对信号量的删除、设置或读信号量的值等操作。有关 semctl 的操作命令,由于较多和比较复杂,这里不再深入讨论。

上面介绍了有关 IPC 的 3 种通信机制。这 3 种机制使用消息和共享存储区方式使多个进程使用同一介质(消息队列或共享存储区)进行通信。这优于管道等通信方式。但是,由于各系统调用中使用的关键字的语义很难扩展到一个网络上(不同的机器上同一关键字可以描述不同的对象),因此 IPC 仍是属于单一机器环境下的通信机构。

## 6.6 Linux 存储管理

存储管理部分还需要对物理内存进行有效的分配和释放。同时内存是一种有限的资源,无法容下全部活动进程。所以存储管理系统必须决定哪个进程的哪个部分应该放在内存,并管理那些不在内存又属于同一进程虚空间的部分。因此 Linux 系统必须解决为进程分配内存空间、进行内存扩充、完成由虚存到物理存储器的地址变换以及内存信息保护与共享等问题。

Linux 采用请求调页策略进行存储器管理。早期的 UNIX 系统只采用交换技术进行主存扩充。交换技术与请求调页策略的主要区别在于:交换技术换入换出整个进程(task_struct 结构和共享正文段除外),因此一个进程的大小受物理存储器的限制;而请求调页策略在内存和外存之间来回传递的是存储页而不是整个进程,从而使得进程的地址映射具有了更大的灵活性,且允许进程的大小比可用物理存储空间大得多。

### 6.6.1 虚存空间和管理

内存管理和硬件的体系结构相关。本节以 80x86 为例,说明 Linux 的地址空间划分和存储管理的基本思想。

80x86 微处理器内部有一个段页式的内存管理单元(mmu)。其中分段单元由 6 个段寄存器以及由操作系统在内存中构造的段描述符表共同作用,用于将逻辑地址转换为虚拟地址(在 Linux 中也称为线性地址)。

一个逻辑地址由段标识符和段内地址组成,例如,cs:0x1000 表示的逻辑地址处于 cs 段,段内偏移量为 0x1000。

段的详细信息由一个 8 字节的段描述符表示,段描述符说明了该段的起始位置、长度、段的类型特征、特权级别和段的属性。所有的段描述符保存在两个段描述符表中,一个称为全局描述符表(GDT),另一个称做局部描述符表(LDT)。而段寄存器的作用就是作为一个索引(Intel 官方的描述是段选择符)。处理器的分段单元通过段寄存器的内容可以查找到相应的段描述符,并进行权限验证和地址转换。

Linux 实际上只是非常有限地使用 80x86 的分段机制。基本上 Linux 只是为了满足

80x86 的要求而最低限度地使用了分段机制。这么做的原因有两个：第一个原因是分页机制已经足以满足 Linux 对内存管理的需求；第二个原因是 Linux 需要工作在众多的硬件平台上，而 80x86 以外的硬件平台上对分段机制的支持功能并不像 80x86 这么强大和完善。所以忽略 Linux 对分段机制的处理并不会对理解 Linux 的内存管理部分产生太大的影响，下面就不再详细描述了。

80x86 的分页单元由页目录基地址寄存器 CR3 以及由操作系统在内存中构造的页目录表和页表共同作用，用于将分段单元转换后的虚拟地址转换为相应的物理地址。

80x86 的地址空间被分页单元划分成以 4KB 为 1 页的线性数组，页的编码范围为（0～$2^{20}-1$）。分页单元总是以页面为最小的处理单位。转换过程如下：32 位的虚拟地址的值被分为 3 个部分，分别是页目录地址、页表地址和页内地址。分页单元通过 CR3 寄存器找到页目录表的基地址，虚拟地址中的页目录部分作为页目录表的偏移量访问到页目录项，其中的内容就是要访问的页表的基地址，虚拟地址中的页表地址作为页表的偏移量访问到页表项，其中的内容就是要访问的页面的基地址。最后将页表的基地址加上虚拟地址中的页内地址就得到了物理地址。虚拟地址的示意图如图 6.17 所示。

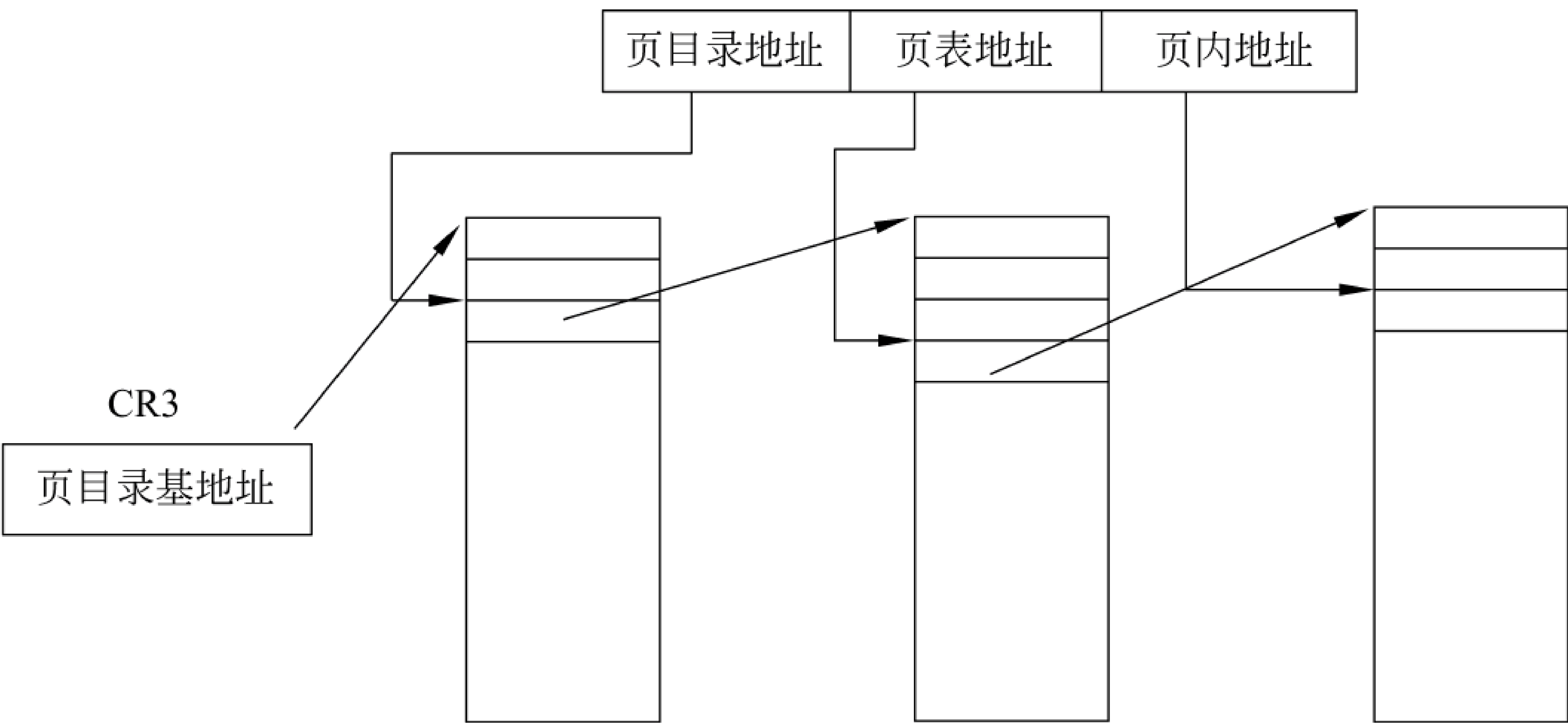



图 6.17　虚拟地址的内容

页表项的内容如图 6.18 所示。

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 第7~0位 | 0 | D | A | 0 | 0 | U/S | RW | P |
| 第15~8位 | 3~0位页面地址 | | | | OS专用 | | | 0 |
| 第23~16位 | 11~4位页面地址 | | | | | | | |
| 第31~24位 | 19~12位页面地址 | | | | | | | |

图 6.18　页表项内容

其中比较重要的有 P(有效)位、D(修改)位、A(访问)位和 RW(读写)位。

第 0 位（最低位）是有效位 P，如果 P＝1，则表示该虚页所对应的内容在内存中，且第 12～31 位所对应的是该虚页的页面基地址。P＝0 时，则该虚页不在内存，硬件将产生相应的出错信息并转由操作系统处理。

第 5 位是访问位 A，指示最近是否有进程访问过该页，该位用于请求调页算法，后面将

进一步介绍。

第6位是修改位D,在内存中,第一次对该页面进行写操作修改该页面内容后,要将该位置1,以便将其换出时写回磁盘保存最新信息。如果D=0,则由于磁盘上保存了该页副本,从而换出时不必写回外存而减少不必要的I/O操作。

Linux中每个进程拥有自己的页目录和页表,在进程调度的时候调度过程会将进程的页目录的地址保存到CR3寄存器,这样就切换了进程的整个地址空间。所以每个进程都拥有自己独立的3G虚拟地址空间。

### 6.6.2 请求调页技术

当出现缺页异常的时候,Linux的缺页异常处理过程被调用,它通过检查地址所属的虚拟区域的属性判断是否是非法的访问,如果是非法访问,则向进程发送SIGSEGV信号终止用户进程,否则进入请求调页处理过程。

当页表有效位P=0时,由有效位定义可知,进程所要访问的页不在内存中。此时该页会有以下几种情况:(1)该页面是首次被访问,还没有分配过物理页面。(2)该页在外存的文件中,例如可执行文件或其他通过mmap系统调用访问的文件。(3)该页在外存交换区中。针对这3种不同的情况,系统除了要在内存中分配或淘汰相应的页面以调入这些页之外,还要正确区分这些页所在的位置,以便迅速地将所需要的页调入内存。

当整个页表项为0时,说明的是前两种情况,这时Linux需要通过地址所在的虚拟区域的属性来判断。如果虚拟区域对应一个磁盘文件,说明是情况(2),这时会为其分配内存,并将磁盘文件的相应内容读到分配到的内存中;如果区域不对应于一个磁盘文件,说明是情况(1),这时候根据请求的是读操作还是写操作有不同的处理:

(1) 写操作,没有优化的手段,直接为其分配空间。

(2) 读操作,因为是第一次访问,这段地址空间的内容应该全部是0,所以Linux并不为其分配物理内存,而是把一个在内核初始化时静态分配的特殊的页面映射到这个页面地址(这个特殊的页面称为零页,它的内容全部是0),同时把页表项设置为只读的。这样当下次进程试图对其进行写操作的时候会触发写时复制机制,直到那时候才为其真正分配物理内存。

写时复制机制在描述进程创建过程的时候就提到过。这里再详细描述一下写时复制是怎么对进程创建起作用的。这里的重点是进程创建的时候父进程的页被设置为和子进程共享,且将页的保护位设置为只读的,而不是为子进程分配新的页面并复制父进程的页面的内容。当父进程或子进程试图写这个页面的时候,缺页异常处理过程会判断出该页面是共享的,并为其重新分配页面并复制内容,并且将页的保护标志设置为可写的,下次再对该页面进行写操作就不会再次触发访问异常。因为子进程在创建以后往往马上调用exec系统调用运行一个新的程序,而放弃父进程的地址空间,这样写时复制避免了很多不必要的复制工作,从而节省了时间。

下面继续请求调页的描述,当页表项不为空的时候,说明访问的页在交换区中。这时候页表项的内容就是该页在交换区中的索引。通过这个索引Linux就可以找到保存在交换区的页面内容,并将其复制到新分配的页面。

Linux可以使用多个交换设备,系统为每个交换设备定义了一个swap_info_struct结

构，该结构描述了此交换设备的设备号、大小和优先级等信息。Linux 按照优先级顺序依次访问交换设备，只有高优先级的交换设备用完了的时候才会开始使用低优先级的交换设备。所以页表项中交换区索引实际上包含了交换设备的索引号和交换设备内页面索引号两个部分。

**1. 交换缓冲**

因为 Linux 在某些情况下页面是在进程间共享的，当一个进程的页面被换出到交换设备的时候，并没有简单的方法可以找到所有共享该页的进程。所以 Linux 把一个共享页面交换出去的时候，并不立刻释放该页面，而是把该页面保存到交换缓冲区中，只有在所有该页面的引用被交换出去后，才有机会释放该页面。所以请求调用的调入过程中，换入过程首先需要检查该页是否在交换缓冲区中，如果是的话，就没有必要启动不必要的 I/O 操作进行换入了。

利用上述几种数据结构可将请求调页时的基本处理过程描述如下：

当 P=0，即发生缺页时，核心先检查缺页原因，如果是已经被换出到交换分区的情况，检查交换缓冲区。如果交换缓冲区中存在一个对应的页面，则不必启动交换设备或交换文件，而直接从交换缓冲区中移出该页即可。图 6.19 给出了该调入过程的流程图。

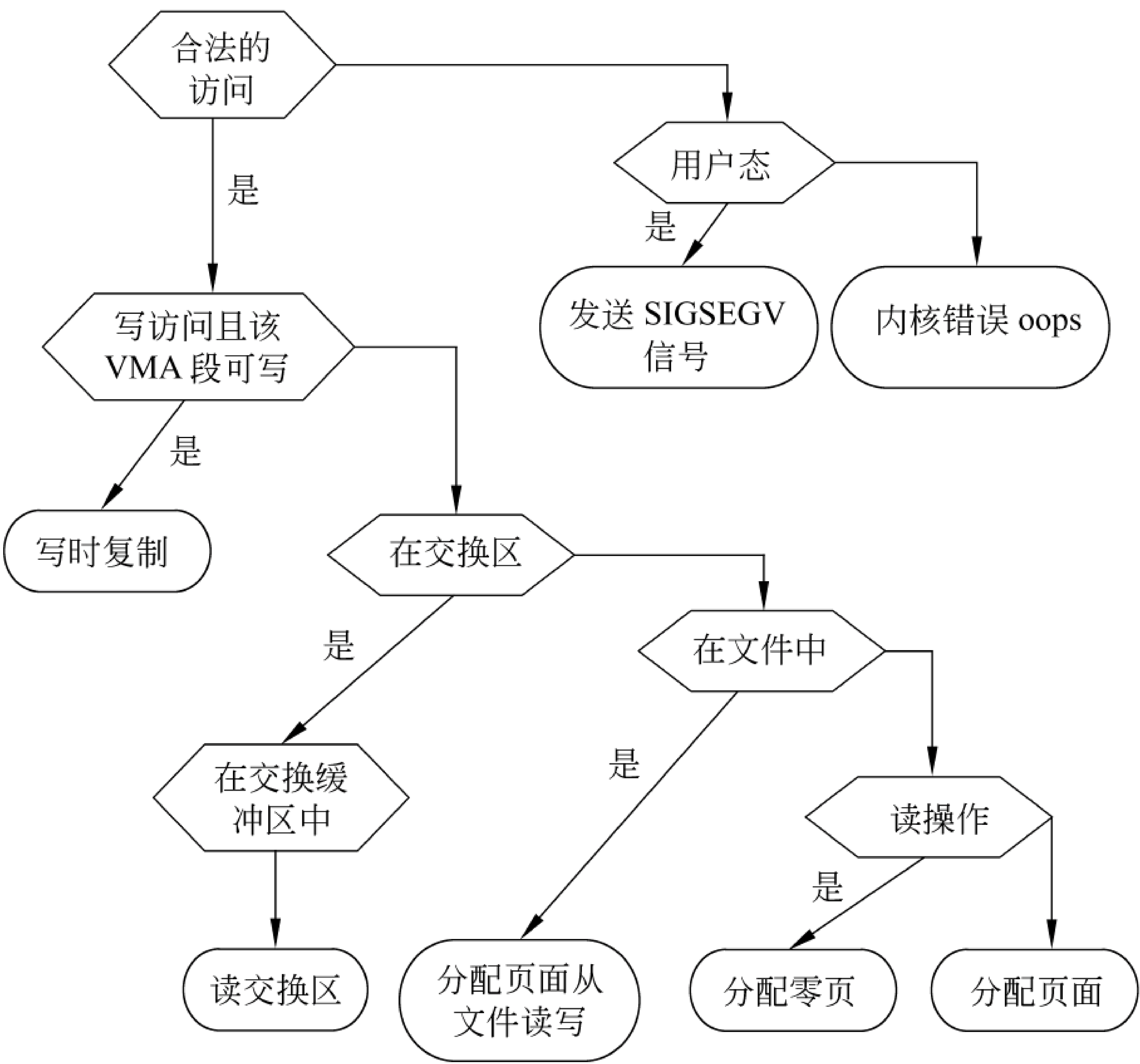



图 6.19 请求调页的调入基本处理过程

例如，当进程访问虚拟地址 1493K 时，页表项中有效位 P=0 从而发生缺页。此时，由磁盘块描述项得到该虚拟地址(1493K)所对应的外存页面在逻辑设备号为 1、块号为 2743 处；且由页表项可知，在该页换出内存前所对应的内存页面号为 794。然后，由图 6.20 中的处理流程，首先根据页面号 794 检查交换缓冲区，发现页面 794 中的数据或程序还在缓冲区中。从而只需从缓冲区中把 794 号页移到内存而不必启动外部设备。

**2. 页面换出过程**

上述过程只是请求调页中的调入过程。Linux将未被使用的物理页作为系统的缓冲区和块设备的缓冲区,如果内存中已无足够的页面存放调入页,Linux首先通过减少各种缓冲区的大小来满足进程的需要。如果这样仍然不够,系统必须淘汰相应的内存页面以存放刚调入的页。考虑到效率,Linux在两种情况下进行页面的换出工作。一个是在分配内存的时候发现空闲内存低于某个极限值时,另一个是使用kswapd核心线程每10秒1次周期性地换出内存。

第一种情况下换出操作的统一入口是try_to_free_pages()函数。首先检查缓冲区,缓冲区按照最近是否访问过分为两个链表,一个是最近访问过的,称为active_list;另一个是最近没有访问过的,称为inactive_list。核心从active_list的队尾向前检查每一个页面,如果该页的访问标志为真,则将其移动到active_list的队列表头,否则将其加入inactive_list的表头。随后从inactive_list的队尾向前检查,如果该页没有被对象引用,则将其释放,否则将其移动到inactive_list的表头。注意这里页面的释放不会涉及交换分区的操作,因为这些缓冲区的内容都对应磁盘上的文件,不需要浪费交换分区的空间。

如果通过释放部分缓冲区仍然不能满足要求,系统会遍历各进程的地址空间,检查页面对应的页表项的访问标志,同样如果访问标志不为真,认为最近没有访问过,从而释放其空间,在这种情况下,因为页面不对应磁盘上的文件,需要在交换分区为其申请空间,并将交换出去的页面保存到交换分区中。

Linux第二种情况下的换出操作的入口是balance_pgdat()函数,其换出操作的过程和try_to_free_pages()函数类似。

由此可见,Linux的内存淘汰策略是近似的最近最少使用(LRU)算法。

**3. 反向映射(reverse mapping)**

在页面换出过程中,需要查找所有关联了该物理页面的页表项,并逐一更新这些页表项,为了快速定位引用了某个物理页面的所有页表项,Linux 2.6中采用了反向映射机制。这种机制的基本思想就是建立物理页面和所有映射了该物理页面的页表项之间的关联,操作系统为此设置了反向映射链表,链表上的节点应用了该物理页面所对应的虚拟内存区域(vm_area_struct结构),虚拟内存区域通过内存描述符(mm_struct结构)找到页全局目录,从而找到相应的页表项,这种方法能节省内存空间,并使遍历反向映射链表时消耗时间减少,从而在很大程度上减少了操作系统在页面回收上所占用的CPU时间。

## 本章小结

本章从Linux中进程的概念出发,介绍了Linux中进程动、静两方面的特性以及描述方法,然后,介绍了Linux的进程创建、调度、执行和撤销等的控制手段、方法以及相应的用户接口。除此之外,还讨论了Linux的进程通信部分,包括用于同步与互斥等控制用的低级通信以及大量传递信息的IPC机制。最后,介绍了与进程管理息息相关的存储管理部分。

对于进程的概念来说,本章重点强调了进程是在进程上下文中执行这个概念。为了深入掌握这一概念,首先必须对什么是进程上下文有一个清楚的认识。本章重点介绍了进程上下文的各个组成部分以及进程上下文的切换方法。

除了进程 0 之外，Linux 的所有进程都是由父进程创建生成的。在父进程创建生成子进程时，系统要为子进程生成自己的上下文，这就需要调用 6.6 节所述的存储管理程序为其分配 task_struct 结构和进程页表等。如果内存中无足够的空间，或超出了用户资源限额，则无法生成子进程。在子进程创建成功之后，马上就看到了子进程和父进程并发执行，抢占处理机的现象。至于谁先抢得处理机，要依靠调度程序决定。Linux 的调度策略是基于优先级的，即选取优先级最高者(优先数最小者）占据处理机。但是，优先级的计算又是与时间片有关的。因此 Linux 的调度是基于优先级加时间片的。

另外，Linux 系统没有作业的概念，用户必须在一个进程的控制之下和系统进行会话，所输入的数据也必须先送到内存工作区之后，再由进程决定是否写入外存文件系统保存。

进程通信一直是早期 UNIX 系统较弱的一部分，但 IPC 机构弥补了这一点。Linux 的 IPC 提供了消息、信号量及共享存储区等大量信息传递方法。不过，IPC 机构只能在一台机器内使用，不能扩展到网络通信上，这是它最大的弱点。

Linux 的存储管理策略是请求调页。请求调页系统只把进程的部分页面放入内存，其他大部分页面在需要时再请求调入。因此，进程大小不再受内存大小的限制。

系统将一个进程的虚拟空间分为不同的虚拟区域，每个区对应一个 vm_area_struct 结构，用户进程通过 mmap 和 munmap 系统调用申请和释放虚拟区域。

## 习　　题

6.1　简述 Linux 系统进程的概念。

6.2　Linux 进程上下文由哪几部分组成？为什么说核心程序不是进程上下文的一部分？进程页表也在核心区，它们也不是进程上下文的一部分吗？

6.3　假定在用户态下执行的某个进程用完了它的时间片，由于时钟中断的原因，核心调度一个新进程去执行。请形式化地描述出新、旧进程的上下文切换过程。

6.4　Linux 的调度策略是什么？调度时应该封锁中断吗？如果不封锁，会发生什么问题？

6.5　试述进程 0 的作用。

6.6　Linux 在哪几种情况下发生调度？

6.7　编写一个程序，利用 fork 调用创建一个子进程，并让该子进程执行一个可执行文件。

6.8　什么是软中断？

6.9　进程在什么时候处理它接收到的软中断信号？进程接收到软中断信号后放在什么地方？

6.10　Shell 符号＞＞将输出追加到一个指定的文件中，如果指定文件不存在，则该命令创建该文件并将输出写入其中；否则，它打开该文件并在该文件中数据尾部接着写入。编写实现＞＞的 C 语言代码。

6.11　编写一个程序，比较使用共享存储区和消息机制进行数据传输的速度。

6.12　形式化地描述 Linux 中消息机制的通信原理。

6.13　Linux 存储管理策略中交换和请求调页方式有何区别？

6.14　在图 6.20 所示请求调页的调入处理过程中，有可能出现空闲页面链表中的页面内容不同于外存设备上的页面内容的情况，此时应取哪一个页面调入内存？为什么？

6.15　简要总结 Linux 进程管理与存储管理部分的联系。

# 第 7 章　Windows 的进程与内存管理

作为完全的 32 位 Windows 版本，Windows NT 代表了微软公司 Windows 操作系统发展中的主流体系结构。因此，我们将基于 Windows NT 体系结构的微软操作系统，如 Windows NT、Windows 2000、Windows XP 和 Windows Server 2003，作为本书介绍 Windows 操作系统的实例对象。

Windows 的系统服务和应用程序都是以进程的形式驻留在内存中的，处理器通过调度这些进程的执行来完成计算任务。本章以 Windows 为例，介绍进程与内存管理的方法。Windows 是多线程操作系统，处理器调度是以线程为基本单位的。因此，本章在讲解进程的同时，也将介绍 Windows 线程的概念和结构。

## 7.1　Windows NT 的特点及相关的概念

作为 Windows 操作系统的组成部分，我们在讲解进程管理和内存管理时，不可避免地会涉及与 Windows 操作系统体系结构和管理机制相关的知识和概念。为了帮助理解，本节将简单介绍与进程和内存管理相关的 Windows NT 的体系结构和有关的概念。

### 7.1.1　Windows NT 体系结构的特点

微软公司将 Windows NT 作为替代包括 Windows 95 和它之前的各个 Windows 版本的主流操作系统，它在进程和内存管理有关的体系结构设计上有一些特点。

- Windows NT 是支持多处理器的操作系统。
- Windows NT 是完全的 32 位操作系统，设计了 4GB 大小的虚拟地址空间，其中 2GB 的地址空间用作进程的私有空间。
- Windows NT 支持 16 位的 Windows 代码，并为其提供独立的运行空间。
- Windows NT 对访问共享内存的进程有严格的安全限制。
- Windows NT 的系统内存空间只能在核心态被访问。

随着计算机体系结构和 Windows 家族的发展，Windows NT 内核版本也升级到 6.0 版本以上，新的内核版本支持 64 位操作系统，具有更多的新特点：

- 新版本增强了 Active Directory，使用了新的虚拟化和管理，采用了 IIS 7.5。
- 可支持多达 64 个物理处理器或最多 256 个系统的逻辑处理器。
- 可支持 Live Migration（动态迁移），支持虚拟磁盘动态调整容量，及 VM 内存动态配置功能，能以虚拟镜像文件于实体主机上开机。
- 可支持无线广域网，网络驱动程序接口规范为 6.20，支持 AVCHD 摄影机以及通用视频类型 1.1。
- 能支持新的用户模式调度框架。
- 能完整支持 S800、S1600 以及 S3200 数据传输速率的 IEEE 1394b 的火线（IEEE

1394)堆栈。

### 7.1.2 Windows 的管理机制

下面介绍 Windows 的一些管理机制，以帮助理解下面关于进程管理和内存管理的描述。

**1. 核心态（Kernel Mode）和用户态（User Mode）**

为了保证操作系统的稳定性和安全性，Windows 将处理器的运行模式分为核心态和用户态。用户的应用程序运行在用户态，而操作系统的内核代码和设备驱动程序则运行在核心态。处在用户态的应用程序不能直接对操作系统的内核数据直接访问，必要时只能通过操作系统提供的系统调用，将请求转到核心态系统服务。处理器模式也切换到核心态，运行完请求后再返回调用的用户程序，并切换到用户态继续运行。因此，一个用户线程在执行时，往往一部分时间运行在用户态，另一部分时间通过系统调用运行在核心态。

运行于核心态的操作系统服务可以访问所有的系统内存和所有的 CPU 指令，可以利用所有的计算机资源来完成复杂的系统管理。Windows 对运行于用户态的应用所能访问的系统资源有很多限制，从而保护了核心的系统资源不受侵害。

所有运行于核心态的系统服务和设备驱动程序都共享同一系统地址空间，这样可以减少数据交换的中间环节，从而提高系统效率。同时也使得开发操作系统相对于开发普通应用更加复杂，开发人员需要处理对复杂的系统结构和资源的管理。

用户态进程拥有自己独立的虚拟地址空间，它不能访问系统地址空间中的数据，也不能直接访问其他用户进程的地址空间。这种设计将进程执行错误所引起的损害限制在出错进程内，保证了操作系统的和其他应用运行的稳定性。

**2. Windows 操作系统的体系结构**

Windows 操作系统是由运行在用户态和核心态的一些构件组成的，一般将运行于核心态的构件称为核心系统服务，而将运行于用户态的构件称为用户进程。

Windows 的用户进程一般包括以下几种：

- 操作系统支持进程。如用于用户登录的进程、会话管理进程等。它们虽然是操作系统中必不可少的一部分，但它们运行在用户态，用于管理用户和操作系统的会话。
- 服务进程。许多基于操作的应用服务器，如 SQL Server 和 Exchange Server 等，用于提供各种应用服务，也运行在用户态。
- 应用程序。所有的用户应用程序都运行在用户态。
- 环境子系统服务进程。为了支持多种操作系统应用的运行，如 Windows 16 位和 32 位应用、MS DOS 应用、POSIX 32 位应用和 OS/2 32 位应用，Windows 包含了多个环境子系统进程，为相应的程序提供运行环境。当应用程序需要进行系统调用时，首先需要通过对相应的子系统动态链接库的调用，将相关的调用转换成对标准的系统服务的调用，再切换到核心态运行。

Windows 的核心系统服务一般包括以下几种：

- Windows 执行体。它是运行在核心态的系统服务，用于管理进程和线程，管理内存，管理设备，提供系统安全、网络以及进程间通信等服务。
- Windows 内核。它为执行体提供底层系统服务，管理线程调度、中断和意外处理、多

处理器同步等。

- 设备驱动程序。它运行在核心态，管理硬件设备和处理 I/O 请求。
- 硬件抽象层。它对不同的计算机环境(主要是主板上的硬件)提供标准的系统封装，使得其他的系统服务在设计时实现和硬件无关。
- 窗口和图形系统。为了实现高效的用户交互，Windows 的窗口管理和图形功能也运行在核心态。

**3. 系统调用、中断和陷阱**

处理器通过陷阱机制捕获当前执行线程，并将控制转到某一特定的处理过程。在陷阱处理之前，系统会记录当前运行线程的核心栈，以便处理完后返回该线程继续执行。

Windows 利用系统服务陷阱来实现用户程序对系统服务调用，当用户线程调用系统服务时会触发系统服务陷阱，并将服务转到系统服务入口，切换到核心态进行执行。

Windows 利用中断陷阱机制来管理硬件设备。通过设备驱动程序设置硬件中断陷阱，当进行 I/O 请求时，系统通过硬件中断处理完成设定的操作。操作系统内核还通过设置软中断陷阱来进行启动线程调度、超时处理、进行非同步的 I/O 操作、非同步调用其他线程的功能等。

Windows 利用意外陷阱机制来管理系统的出错状态，当发生意外处理事件时，系统会根据意外事件的条件转到特定的意外处理例程。

**4. 利用对象来共享系统资源**

Windows 操作系统服务定义了各种系统数据结构，Windows 对只在系统服务内部使用的数据往往用简单的结构数据来表示，以便提高系统效率。对于需要被用户态程序访问的系统数据，Windows 都是用对象来表示的，如进程、线程、文件和事件等。

对象的特点是必须通过对象服务来访问和修改对象封装的数据。系统服务在为用户态的进程提供访问接口时，可以利用对象的数据封装有效地防止破坏性的操作。在用户态的进程访问系统对象时，往往使用对象句柄。对象句柄是可以引用对象的间接指针(以免直接访问系统数据结构)。我们在下面讲到的关于进程结构时，会遇到很多与对象和对象句柄相关的概念。

在严格意义上，Windows 不是一个面向对象实现的操作系统，大部分操作系统的代码是使用 C 语言来编写的，但 Windows 操作系统在系统设计上采用了面向对象的一些思想，这样有利于在提高系统效率的同时增强系统的可移植性。

**5. 本地过程调用**

本地过程调用是 Windows 操作系统为系统服务进程间进行高速消息传递设计的通信机制，它不提供用户态的调用接口。调用的双方分为服务进程和客户进程，它们通过端口对象来进行通信。为了处理多个调用请求，调用首先在调用队列中排队，当接受请求后，会生成客户通信端口和服务通信端口来完成通信。

## 7.2 Windows 进程和线程

构建 Windows 的系统服务和所有的应用程序都是以进程的形式驻留在内存中的，由处理器调度执行来完成设定的计算功能。每一个进程都包含一个或多个线程，Windows 处理

器调度的对象是线程。本节介绍 Windows 进程和线程的结构以及它们之间的关系。

### 7.2.1 Windows 的进程和线程的定义

一个 Windows 进程有自己独立的虚拟地址空间,用以保护私有的执行代码和数据不被其他的进程破坏。每个进程包含一个或多个线程,运行在一个进程中的线程可以创建新的线程和新的进程。

在 Windows 操作系统中,处理器调度的对象是线程,而进程为线程的运行提供资源和上下文环境,保证所属的线程在进程的虚拟地址空间范围内运行。

一个 Windows 进程包含以下信息:

- 唯一的进程标识。
- 一个独立的虚拟地址空间。
- 映射到进程虚拟地址空间的执行代码和数据。
- 访问各种系统资源的对象句柄列表。
- 安全上下文定义来说明与进程相关的用户、安全信息和访问特权设定。
- 至少包含一个可执行的线程。

一个 Windows 线程包含以下信息:

- 唯一的线程标识。
- CPU 寄存器的状态数据,用以表示处理器的状态。
- 两个线程栈,一个在用户态执行时使用,另一个用在核心态执行时使用。
- 一个供子系统、运行库和动态链接库使用的线程本地存储空间。

### 7.2.2 进程和线程的关联

Windows 的进程和线程是紧密相关的,系统通过创建进程来为线程提供必要的上下文环境,如内存、资源对象等,进程以执行进程块的形式驻留在内存中。系统通过创建线程来运行具体的程序,同时提供处理器调度所需要的信息,线程以执行线程块的形式驻留在内存中。一个运行的线程可以在同一进程中创建新的线程来运行新的功能,也可以通过创建新的进程来启动新程序的运行环境。

如图 7.1 所示,构成进程的数据由分别驻留在系统地址空间中的核心进程块和驻留在进程地址空间中的进程环境块构成,类似地,构成线程的数据也由分别驻留在系统地址空间中的核心线程块和驻留在进程地址空间中的线程环境块构成。进程通过核心进程块中指向核心线程块的指针来访问该进程所属的线程,还通过进程环境块为运行在用户态的服务访问进程提供接口。同样,线程也通过驻留在进程地址空间中的线程环境块为运行在用户态的服务访问线程提供接口。

线程通过线程环境块中的指针指向它属于的进程的进程环境块,因此线程调度器可以通过线程访问进程环境提供的上下文信息。

### 7.2.3 Windows 进程的结构

如表 7.1 所示,Windows 将表示进程的数据结构统称为执行进程块,它提供了操作系统管理进程所需要的基本信息。执行进程块包括本进程和父进程的标识、进程使用系统内

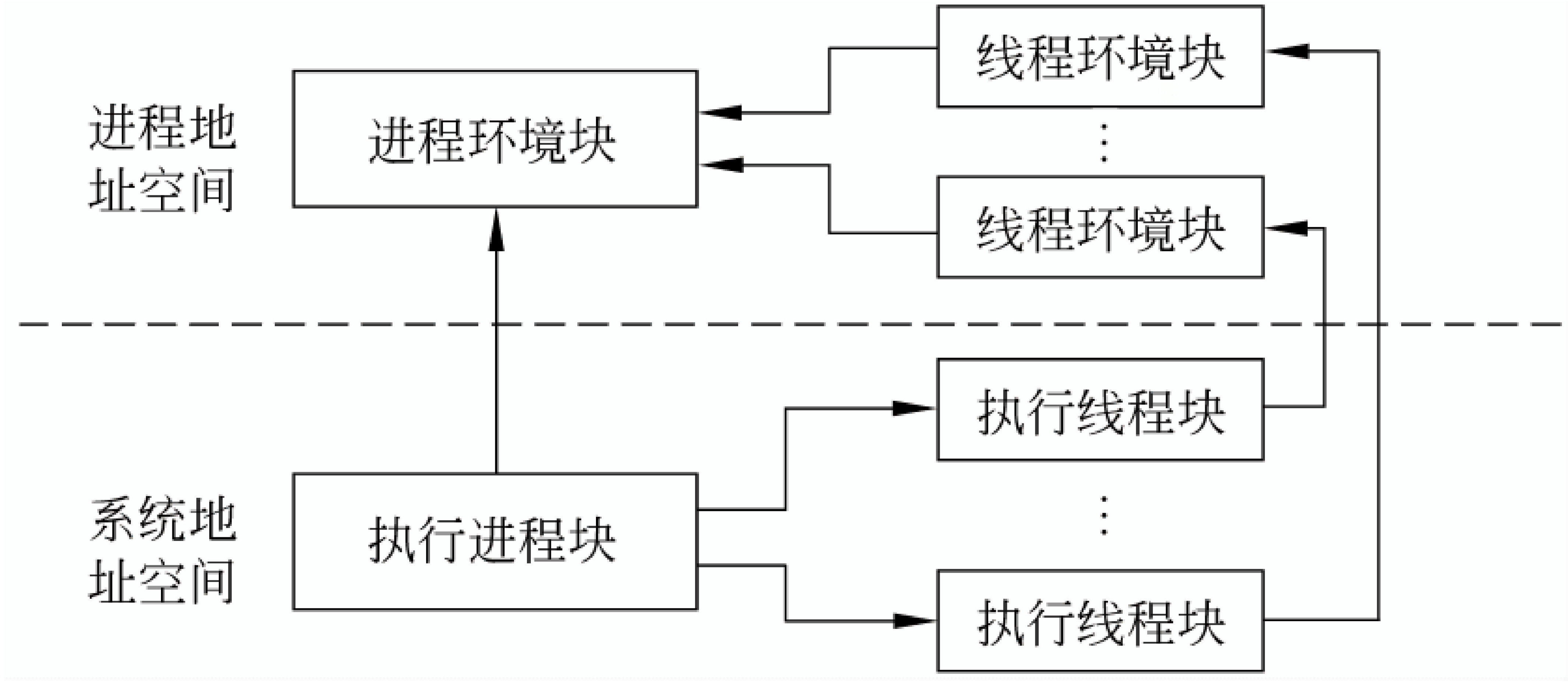



图 7.1　Windows 进程和线程的关联

存的配额、进程管理虚拟内存的信息、进程所使用的对象句柄、进程访问的安全性描述以及供 Windows 子系统访问的进程信息等。执行进程块还有一部分数据被称为核心进程块，它包含了 Windows 内核调度所属线程需要的基本信息，如时间片、核心栈和进程基准优先级等。除此之外，还有一部分称作进程环境块的数据驻留在进程地址空间中，提供映像调入器、堆管理器和其他运行在用户态的动态链接库所需要的进程信息，如程序映像的基地址、用户栈信息和线程的局部存储空间。

**表 7.1　Windows 进程的数据结构**

| 执行进程块的数据项 | 功　　能 |
|---|---|
| 进程标识 | 唯一的进程标识号、父进程标识号和运行的镜像文件(Image)名称 |
| 系统资源配额 | 对该进程所使用系统内存池以及分页文件的配额限制 |
| 虚拟内存管理 | 用来描述进程的哪些虚拟地址空间已被占用、哪些空间可以使用，以及进程虚拟内存管理的状态信息 |
| 工作集信息 | 该进程的虚拟地址空间中驻留在物理内存中的页面的集合 |
| 意外本地过程调用端口 | 当进程所属的线程出现意外时，进程管理器会通过本地过程调用发送意外信息到该进程，通过该端口接收意外信息进行相应的处理 |
| 调试本地过程调用端口 | 当进程所属的线程触发调试事件时，进程管理器会通过本地过程调用发送调试消息到该进程，通过该端口接收调试消息进行相应的处理 |
| 访问安全描述 | 访问该进程的安全设置 |
| 对象句柄表 | 指向该进程的所有对象句柄 |
| Windows 子系统进程信息 | Windows 子系统调用该进程所需要的进程信息 |
| 核心进程块 | 包含了 Windows 内核调度该进程的所属线程所需要的基本信息，如分配给该进程的处理器时间、时间片大小、核心栈信息、进程基准优先级以及进程状态等 |
| 进程环境块 | 驻留在进程地址空间中，提供映像调入器、堆管理器和其他运行在用户态的动态链接库所需要的进程信息，如程序映像的基地址、用户栈信息和线程的局部存储空间 |

### 7.2.4　Windows 线程的结构

如表 7.2 所示，在 Windows 将表示线程的数据结构统称为执行线程块，它为管理线程

提供了线程的基本信息。执行线程块包括该线程所属的进程、线程创建和结束的时间、线程运行的起始例程的地址、线程级别的安全控制和等待处理的I/O请求包列表等。其中一部分数据被称作核心线程块，用来存储用于处理器调度线程的相关信息，如执行时间、优先级和核心栈的地址等。还有一部分被称作线程环境块的线程数据驻留在进程地址空间中，它为调入映像和动态连接库提供上下文信息。

**表 7.2 Windows 线程的数据结构**

| 内 容 | 功 能 |
| --- | --- |
| 线程时间 | 线程创建和结束时间 |
| 所属进程标识 | 所属进程的标识 |
| 起始地址 | 线程起始例程的地址 |
| 访问安全控制 | 线程级别的访问安全控制信息 |
| 局部过程调用信息 | 线程处理局部过程调用的消息标识以及处理消息的地址 |
| I/O 信息 | 等待处理的 I/O 请求包的列表 |
| 核心线程块 | 存储系统进行线程调度和同步的线程信息，如该线程的可用执行时间、核心栈的地址、指向系统服务列表的指针、线程环境块的指针以及与处理器调度相关的信息（如调度优先级、时间配额和空闲处理例程） |
| 线程环境块 | 驻留在进程地址空间，存储用于映像调入器和 Windows 动态链接库所使用的线程上下文信息，如线程的唯一标识、用户栈的地址以及指向所属进程的进程环境块 |

### 7.2.5 Windows 进程和线程的创建

Windows 在启动操作系统时，会创建一些系统进程和线程，为应用程序的运行提供支撑平台。应用程序也是通过创建进程和线程来完成设定的功能的。本节通过分析 Windows 创建进程和线程的过程，介绍进程和线程的管理机制。

**1. 进程的创建过程**

应用程序是通过调用相应的进程创建函数来创建一个新的 Windows 进程的，最常用的进程创建函数是 CreateProcess。

创建进程的过程是由 3 个部分配合完成的：创建进程的系统服务、Windows 子系统和新的进程。CreateProcess 通过调用相关的系统服务来调入需要执行的映像文件并创建进程和初始线程对象，并通过消息通知 Windows 子系统新的进程和线程对象已被创建。Windows 子系统在安装了新的进程和线程后，通知进程管理器执行初始线程。初始线程将新的进程初始化，并开始执行设定的代码。

如图 7.2 所示，Windows 通过 CreateProcess 函数来创建进程的过程如下：

（1）CreateProcess 通过调用进程管理服务找到执行文件的映像，并为其创建区域对象。

（2）创建执行进程对象，包括设置执行进程块，初始化进程的地址空间，初始化核心进程块和进程环境块。

（3）创建初始线程，包括创建执行进程块，设置线程的唯一标识和创建线程环境块。

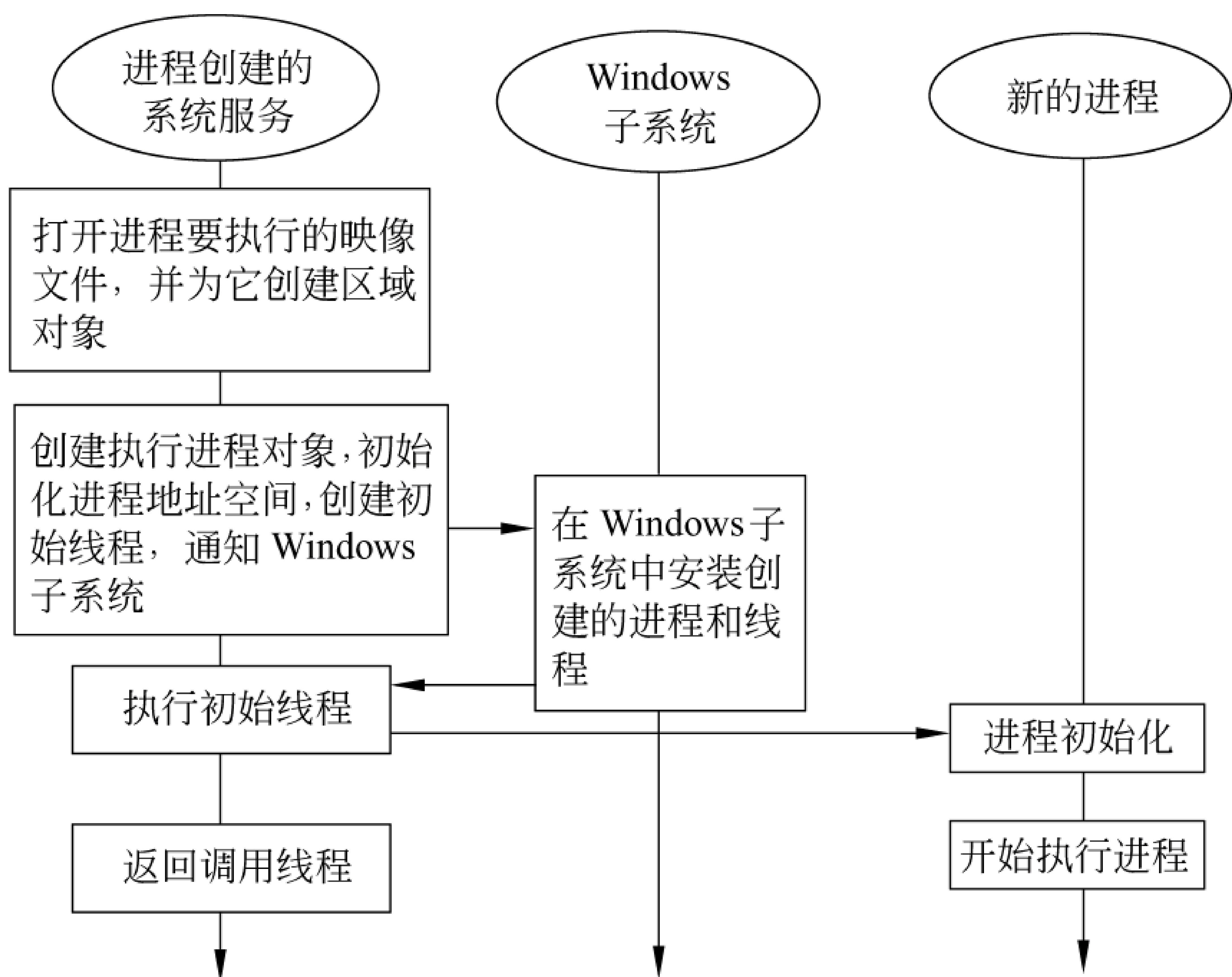



图 7.2　Windows 创建进程的过程

(4) 通过发送消息通知 Windows 子系统新的进程已被创建，子系统将初始线程插入进程的线程列表，并将新的进程加入子系统进程列表。

(5) 将控制传回进程的初始线程，对进程进行初始化。

(6) 在新的进程和线程的上下文内调入相应的动态链接库，开始执行程序。

**2. Windows 线程的创建过程**

应用程序是通过调用相应的线程创建函数来创建一个新的 Windows 线程的，最常用的进程创建函数是 CreateThread。

CreateThread 通过系统服务调用将创建线程的请求传递到位于 Windows 执行体中的进程管理器，它创建线程对象，并调用相应的系统服务初始化核心线程块。Windows 通过调用 CreateThread 创建线程的过程如下：

(1) 在进程的地址空间中为线程创建用户栈，并初始化运行上下文环境。

(2) 初始化线程的线程环境块。

(3) 创建执行线程对象。

(4) 通知 Windows 子系统新线程已被创建，子系统将新线程的线程 ID 插入到相应进程的线程列表中。

(5) 新线程的句柄和标志被返回到调用的线程。

(6) 线程进入调度队列等待执行。

## 7.3　Windows 处理器调度机制

Windows 处理器调度的粒度为线程，Windows 为每一个线程分配调度优先级。调度器根据优先级采用抢占式调度策略，让具有最高优先级的线程首先执行。每一个线程都分配了以时间配额(quantum)为单位的执行时间，通过改变线程的状态来进行线程调度。

### 7.3.1 调度优先级

Windows 在分配处理器时间时，不考虑调度对象属于哪一个具体的进程，不同进程的线程原则上具有同样的调度优先级，优先级的设定更多的是考虑线程要求完成的时间紧迫性。

Windows 内核使用 32 个优先级别来表示线程要求执行的紧迫性，用 0～31 的数字表示。按照优先级的功能不同，它们可以被分为 3 组：

- 16 个实时优先级别(16～31)
- 15 个可变优先级(1～15)
- 1 个系统优先级(0)，为内存页清零线程保留

在应用创建线程时，用户可以用更加形象的优先级描述来设置优先级。当创建进程时，可以赋予以下优先级别：实时、高、高于一般、一般、低于一般和空闲。当创建线程时，在进程的优先级别上进一步赋予以下优先级别：尽量实时、最高、高于一般、一般、低于一般、最低和空闲。这样应用可以用比较形象的方式来设定线程优先级，它和系统的优先级别对应关系如图 7.3 所示。

| 实时 | 高 | 高于一般 | 一般 | 低于一般 | 空闲 |
|---|---|---|---|---|---|
| 36 | 15 | 12 | 10 | 8 | 6 |
| 35 | 14 | 11 | 9 | 7 | 5 |
| 24 | 13 | 10 | 8 | 6 | 4 |
| 23 | 12 | 9 | 7 | 5 | 3 |
| 22 | 11 | 8 | 6 | 4 | 2 |

图 7.3　应用优先级别和系统的优先级别的对应关系

处理器调度时参考两个优先级设置，一个是从当前线程所在的进程的基准优先级，另一个是线程的优先级。一般来讲，应用线程运行在可变优先级别(1～15)的范围内，如果需要进入实时优先级别(16～31)范围来运行，必须取得更高的调度优先级特权。Windows 操作系统只有一个内存页清零线程，它具有最高的调度优先级别(0)，以保证系统内存管理的效率。

### 7.3.2 线程状态

在处理器调度线程执行的过程中，通过改变线程的状态对多个线程进行有效的管理。图 7.4 表示了一个线程在处理器调度过程中的状态变迁。

下面通过表 7.3 对线程状态的含义作进一步的说明。

**表 7.3　线程状态的说明**

| 状　态 | 含　　义 |
|---|---|
| 就绪 | 表示一个线程已经准备就绪，等待运行。调度器查找线程库中处于就绪状态的线程，来决定下一个运行的线程 |
| 预备 | 表示一个线程已经被选择作为下一个运行的线程，如果条件满足，调度器会将上下文环境切换到该线程。但处于预备状态的线程也可能会被转换到就绪状态继续等待 |

| 状　态 | 含　　义 |
| --- | --- |
| 运行 | 当调度器将上下文环境切换到一个进程，该线程就处于运行状态。当分配给线程的时间配额用完，或有更高优先级的线程抢占 CPU，它会让出处理器 |
| 等待 | 当一个线程需要等待必要的系统资源时，会转入等待状态，直到系统资源就绪 |
| 过渡 | 如果一个线程已经准备就绪，但运行它所需要的核心栈暂时被分页调度到磁盘上，它就进入过渡状态，等待核心栈调入内存 |
| 终止 | 当一个线程完成执行后，就进入了终止状态，对象管理器释放相应的执行线程块资源 |
| 已初始化 | 一个线程刚被创建时的内部状态 |

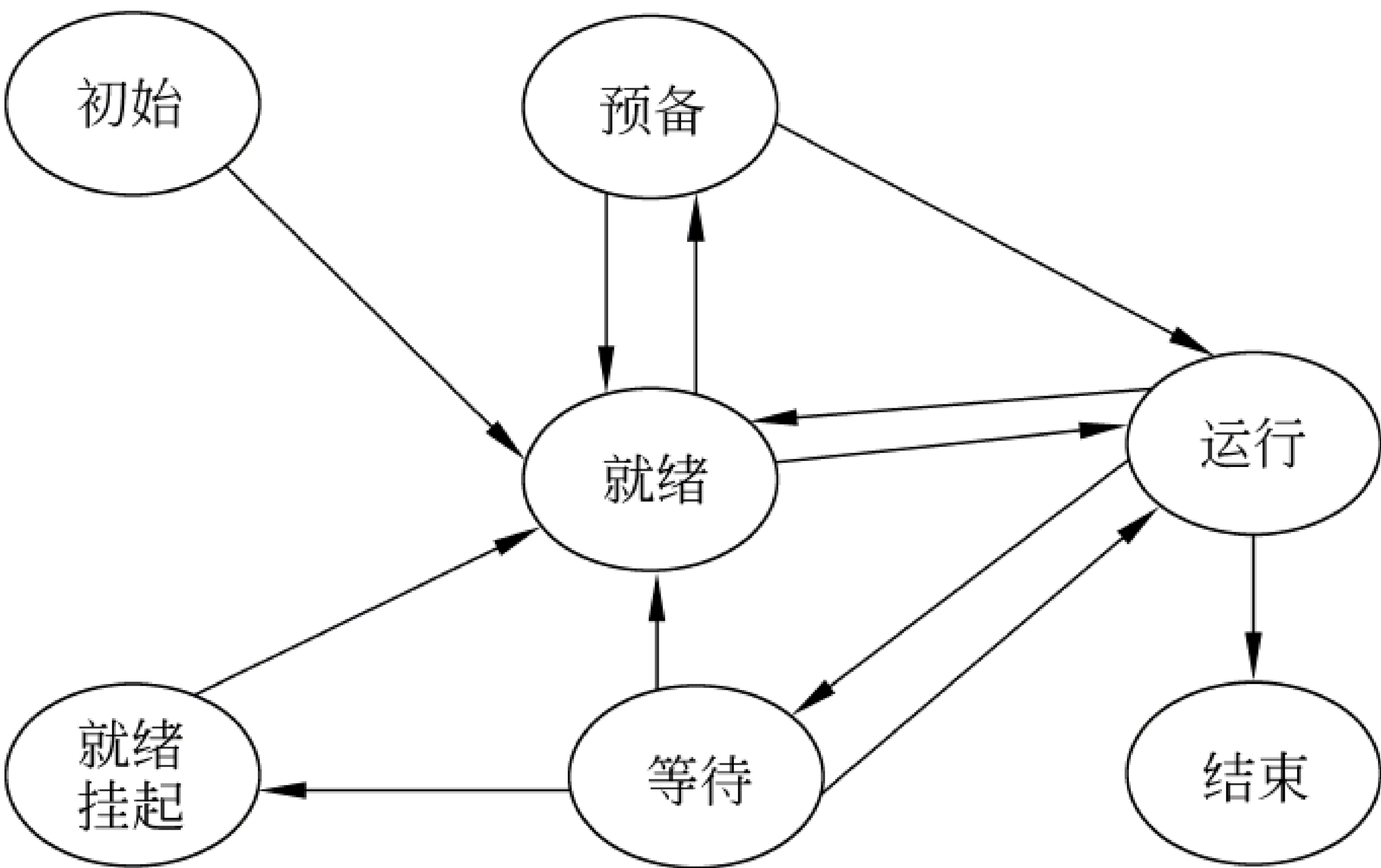



图 7.4　Windows 线程状态转换图

### 7.3.3　线程调度机制

Windows 通过调度数据库来为每一个优先级别的线程维护一个就绪等待队列，当处理器需要调入一个线程运行时，系统会从调度数据库中找到一个具有最高优先级别的就绪线程，并给它分配执行时间。如果等待队列中有线程比正在运行的线程的优先级更高，运行的线程就会保存它运行的上下文环境并进入就绪队列，高优先级的线程恢复它的上下文环境，并进入运行状态。下面对 Windows 线程调度的机制和算法作详细的说明。

**1. 调度数据库**

系统中同时有多个线程存在，而每个处理器在一个时刻只能运行一个线程，Windows 用调度数据库记录处于就绪状态的线程，以便在确定下一个执行的线程时参考。

由于每个线程的优先级别不一样，如图 7.5 所示，Windows 为每一个优先级维护了一个线程队列。调度器找到优先级最高的线程队列，从队列头找到下一个运行的就绪线程。

**2. 时间配额**

当一个线程进入运行状态时，它获得了一个可以运行的时间配额。线程在核心线程块中都记录了当前的时间配额值，每过一个时钟周期，该值就会减 1，当该值变为零时，表示时间配额已经用完。

当分配给该线程的时间配额用完时，调度器会查找调度数据库看是否有就绪的线程在等待执行。如果有等待的就绪线程，调度器会将正在执行的线程转入等待或就绪状态，调入下一个具有最高优先级的线程进行运行。如果调度数据库中没有等待运行的就绪线程，调度器就再分配一个时间片让该线程继续运行。

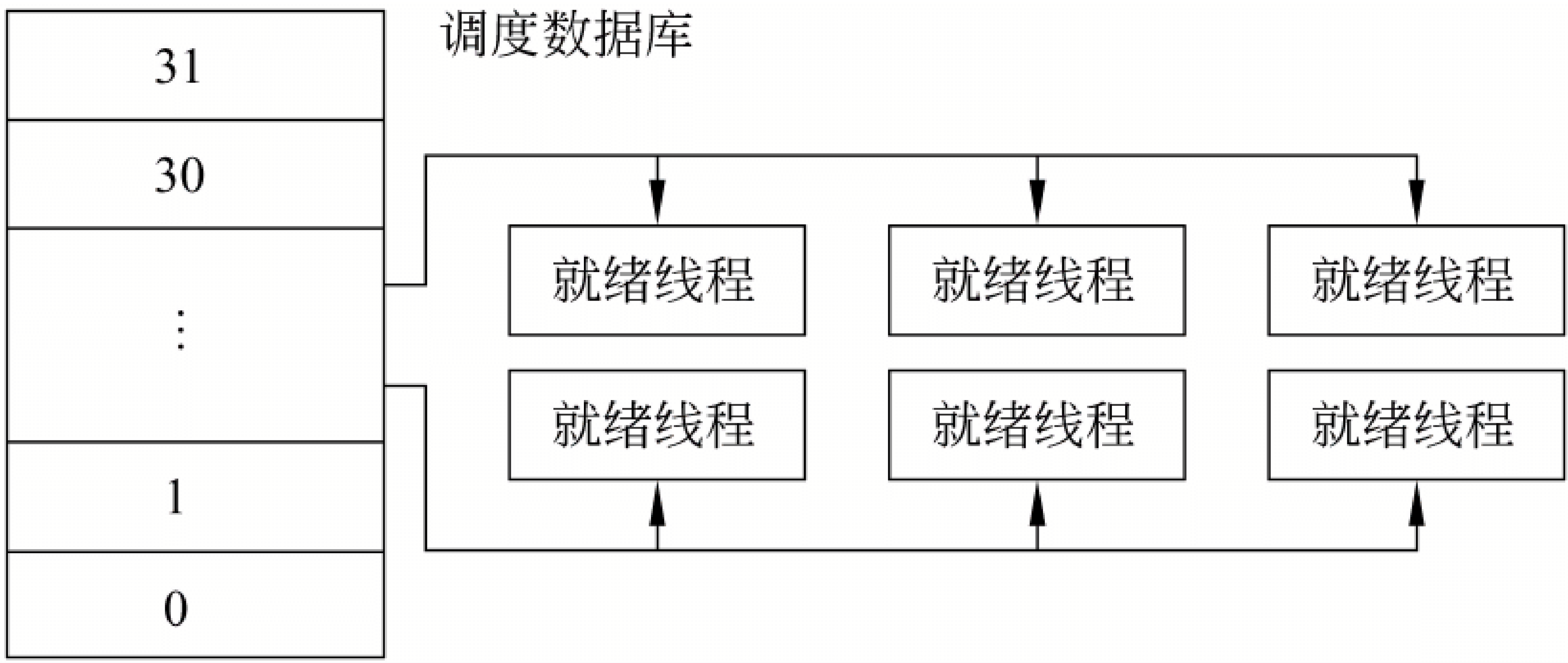



图 7.5　调度数据库的不同优先级就绪线程队列

PC 版的 Windows 操作系统和服务器版的 Windows 分配给线程的运行时间片的长短有所不同。PC 版的 Windows 分配的时间片较短，以免某些线程占用太多的运行时间，影响用户与计算机之间的交互。服务器版的 Windows 分配的时间片较长，可以给后台线程更多的时间完成服务请求，避免频繁的 CPU 切换。

**3. 调度算法**

Windows 采用基于优先级的抢占式调度算法，在一个具有较低优先级别的线程正在运行时，如果有一个具有较高优先级别的线程进入就绪等待队列，或一个具有较高优先级别的线程的等待结束时，就可以抢占处理器优先执行。为了不让被抢先的线程等待太长的时间，调度器会将它移到等待就绪队列的队头。

当一个正在运行的线程需要等待某一个对象时，包括事件、互斥的状态解锁等，会主动地让出处理器而进入等待状态。系统为每一个需要等待的对象维护了一个等待队列，让出处理器的线程会进入队尾等待，处在调度数据库中具有最高优先级的线程将被分配执行时间，并开始运行。

当一个线程的时间配额用完时，如果等待队列中有就绪线程等待，调度器会将当前线程移到就绪等待队列的队尾，而为具有最高优先级的线程分配运行时间，并开始运行。

当一个线程完成所有的代码运行，调度器会将它的状态设为结束。系统会将它从所属进程的线程表中去掉，并释放它使用的内存和系统资源对象。如果调度数据库中没有等待运行的就绪线程，Windows 会调入空闲线程。

**4. 上下文切换**

处理器调入和调出一个线程时，是通过切换上下文来实现的。当系统调出一个正在运行的线程时，需要保存的线程上下文信息为运行指令指针、用户栈和核心栈指针以及线程所在进程的虚拟地址空间指针。线程的核心栈用来完成上下文的切换，调度器将调出线程的上下文环境信息压入该线程的核心栈，并将栈指针保存到该线程的核心线程块中。

在调入一个线程执行时，核心栈指针指向调入线程的核心栈，并恢复调入线程的执行上下文。如果调入线程所在的进程需要改变，处理器会将相关的地址寄存器设置到新进程的虚拟地址空间。处理器再将控制转到调入线程的运行指令指针，开始运行该线程。

## 7.4　Windows 的内存管理

Windows 的存储管理主要包括内存管理和外存管理，本节主要讲述 Windows 内存管理的机制，在 11 章中将结合文件系统介绍 Windows 的外存管理。

不同版本的 Windows 操作系统支持的最大物理内存是不一样的，如 Windows XP 最大支持 4GB 物理内存，Windows Server 2003 最大可以支持 64GB 的物理内存，Windows Server 2008 最大可以支持 2TB 的物理内存。32 位的 Windows 操作系统定义了大小为 4GB 的虚拟内存空间，以便使系统服务和应用可以在固定大小的内存视图上操作。内存管理器将虚拟地址转换到实际的物理内存地址，如果需要访问的数据不在物理内存中，系统通过换页机制将它调入内存。

### 7.4.1 内存管理器

Windows 通过内存管理器来管理内存，它的一个重要功能就是将进程的虚拟地址映射到具体的物理内存地址。当系统可用的物理内存不够时，内存管理器会将一部分驻留在物理内存中的数据通过分页机制调出到磁盘上，当需要时再将它调入物理内存。一般的计算机实际提供的物理内存都小于应用需要的内存，因此，进程一般只有一部分虚拟地址空间中的数据驻留在物理内存中，这一部分虚拟地址空间被称为工作集。

内存管理器是 Windows 执行体的一部分，它由一组运行在核心态的系统服务组成，用来分配、释放和管理虚拟内存。它还包含内存访问出错陷阱处理程序，用来处理内存访问时出现的出错事件。

### 7.4.2 内存管理的机制

**1. 页**

内存管理器将虚拟内存空间划分成固定大小的单元——页(page)。页的大小根据具体的计算机体系结构而定，对于 80x86 体系结构的处理器，页大小一般为 4KB。

应用程序一般希望申请到连续的虚拟地址空间，内存管理器通过预留机制来实现这一功能。当线程申请预留一部分内存时，内存管理器对相关的页做预留标记，但在使用之前，并没有将预留的虚拟地址空间映射到实际的物理内存空间。这样可以减少分配内存的时间，在需要使用时再进行地址映射。例如，每个线程的用户态栈就是通过预留机制来申请连续的虚拟缓冲区(默认的大小为 1MB)。

**2. 共享内存**

Windows 提供了在进程间共享内存的机制，共享内存可以理解为同一块物理内存在不同进程空间中的映射。如图 7.6 所示，当两个进程用到同一个动态链接库时，系统只将它调入内存一次，而将它映射到所有使用它的进程空间中。

如果共享内存是只读的，那么这种共享就不会对共享的数据产生影响。但当共享内存是可以写入的，一个线程对共享内存内容的修改会影响其他进程的数据一致性。Windows 采用“复制后写入”的方式来处理这种情况。

如图 7.7 所示，如果进程 2 试图对共享的页进行写入时，内存管理器会给该共享页重新分配另一个物理内存，并将该页的内容复制到新的页中。进行写入操作的进程将该页映射到新的物理内存中，并完成写入操作，而且该页面成为进程 2 的私有页。

**3. 堆管理**

一般来讲，内存申请是以页为单位的(4KB)。当线程需要申请较小的内存块时，如果按照内存页分配机制，内存管理器只能为它申请以页为单位的内存块，会造成了内存资源的浪

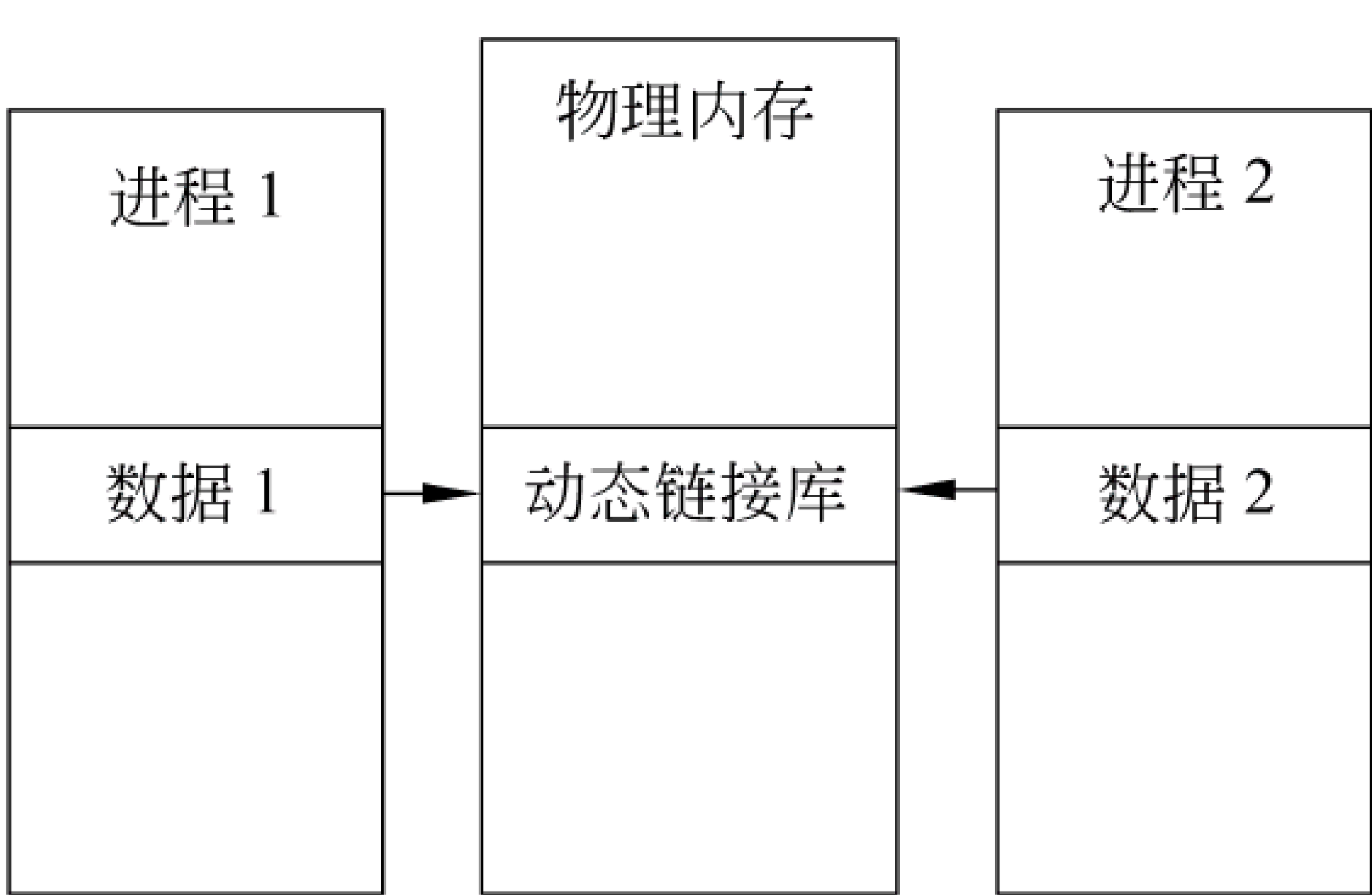



图 7.6　共享内存机制

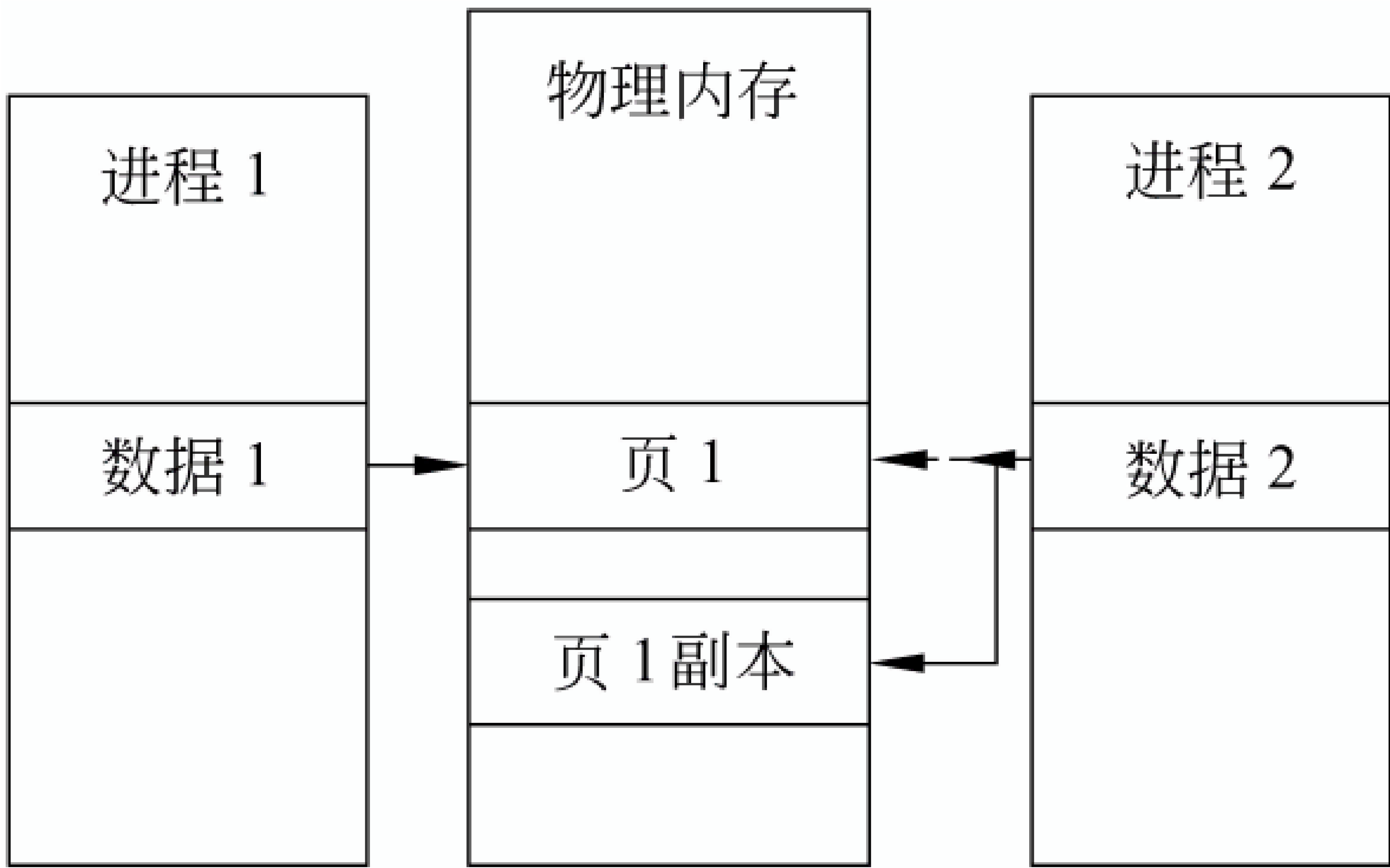



图 7.7　可写入共享内存机制

费。为了解决这一问题，Windows 使用堆管理器来管理小的内存分配。对于 32 位的 Windows 操作系统来说，堆管理器的内存分配粒度可以小到 8B。

每个进程都有一个进程堆，在进程创建时系统分配默认大小为 1MB 的空间作为进程堆空间。在分配的内存大于默认堆的大小时，系统将动态增加进程堆的空间。

**4. 系统内存池**

正如我们在前面提到的，所有的核心态系统服务都共享同一个系统地址空间。内存管理器利用两种大小动态改变的内存池来管理核心态服务的内存分配。

在系统服务中，有一部分的程序和数据需要常驻内存，以保证系统的效率，如中断处理程序等。内存管理器用不分页的内存池来为这些服务分配内存，这部分内存不会被调出到磁盘上。

在操作系统中，还有一部分程序和数据可以被调出到磁盘上，如设备驱动程序等，以便为其他的程序和数据提供物理内存空间。内存管理器用分页内存池来为它们分配内存，在必要时这部分内存会被系统调出到磁盘上。

这两个内存池都驻留在系统的虚拟地址空间的高端部分，线程需要通过在核心态的系统调用来访问这一部分空间。

## 7.5　虚拟地址空间

Windows 通过虚拟地址空间提供了一个内存的逻辑视图，在虚拟地址空间连续的内存在物理分布上并不一定连续。应用程序通过内存管理器将虚拟地址映射到实际的物理内存地址上。

### 7.5.1　虚拟地址空间布局

32 位 Windows 操作系统的虚拟地址空间大小为 4GB。操作系统将低端的一半虚拟地址空间(0x0000 0000 到 0x7FFF FFFF)分配作为进程的私有地址空间，而将高端一半的虚拟地址空间(0x8000 0000 到 0xFFFF FFFF)分配给操作系统内核作为系统地址空间。

如图 7.8 所示，基于 x86 体系结构的 Windows 的虚拟地址空间可以分为几个主要的部分。从 0x0000 0000 到 0x7FFF FFFF 的 2GB 的低端地址空间为进程的私有空间，用来存储进程的应用代码、全局变量和线程堆等数据。

| 地址 | 区域 |
|---|---|
| 0x0000 0000 | 进程私有地址空间（应用代码、全局变量、线程堆） |
| 0x8000 0000 | 内核及执行体、硬件抽象层、引导驱动 |
| 0xC000 0000 | 进程页表和工作集 |
| 0xC080 0000 | 系统缓存、系统内存池 |
| 0xFFFF FFFF | |

图 7.8　32 位 Windows 的虚拟地址空间布局

高端一半的虚拟地址空间(0x8000,0000 到 0xFFFF,FFFF)为系统地址空间。在系统空间地址的低段部分(0x8000,0000 到 0xBFFF,FFFF)驻留了系统内核及执行体、硬件抽象层、引导驱动等系统服务。在系统地址空间的中段(0xC000,0000 到 0xC07F,FFFF)驻留了描述进程内存使用情况的页表和工作集等信息。在系统地址空间的高段(0xC080,0000 到 0xFFFF,FFFF)用于系统高速缓存、系统内存池等系统资源。

表 7.4 列出了 32 位 Windows 操作系统的虚拟地址空间的详细分布。

**表 7.4　基于 32 位 x86 体系结构的 Windows 的虚拟地址布局**

| 地址范围 | 大小 | 功　　能 |
|---|---|---|
| 0x0000,0000～0x7FFF,FFFF | 2GB | 进程的私有地址空间(程序代码、全局变量和线程栈等) |
| 0x8000,0000～0x9FFF,FFFF | 512MB | 系统内核(NTLDR，HAL)和引导驱动 |
| 0xA000,0000～0xA2FF,FFFF | 48MB | 系统映射视图或会话空间 |
| 0xA300,0000～0xA3FF,FFFF | 16MB | 终端服务的系统映射视图 |
| 0xA400,0000～0xBFFF,FFFF | 448MB | 附加系统页表入口或附加系统高速缓存 |
| 0xC000,0000～0xC03F,FFFF | 4MB | 进程页表 |
| 0xC040,0000～0xC07F,FFFF | 4MB | 工作集链表 |
| 0xC080,0000～0xC0BF,FFFF | 4MB | 未使用 |
| 0xC0C0,0000～0xC0FF,FFFF | 4MB | 系统工作集链表 |
| 0xC100,0000～0xE0FF,FFFF | 512MB | 系统高速缓存 |
| 0xE100,0000～0xEAFF,FFFF | 160MB | 分页缓冲池 |
| 0xEB00,0000～0xFFBD,FFFF | 331MB | 系统页表入口和非分页缓冲池 |
| 0xFFBE,0000～0xFFFF,FFFF | 4MB | 故障处理和硬件抽象层(HAL)结构 |

### 7.5.2　虚拟地址转换

系统服务和应用程序是通过虚拟地址来操作内存的，访问之前需要将虚拟地址映射到实际的物理内存地址，内存管理器利用页表来进行地址转换。页表存储在系统地址空间中，每个虚拟地址都和一个页表入口相关。

如图 7.9 所示，内存管理器将虚拟地址映射到物理地址时，首先通过页表查到该虚拟地址的页表入口，再通过页表入口查到该地址对应的物理内存地址。在虚拟地址空间连续的

内存，它们所对应的物理地址往往是不连续的。

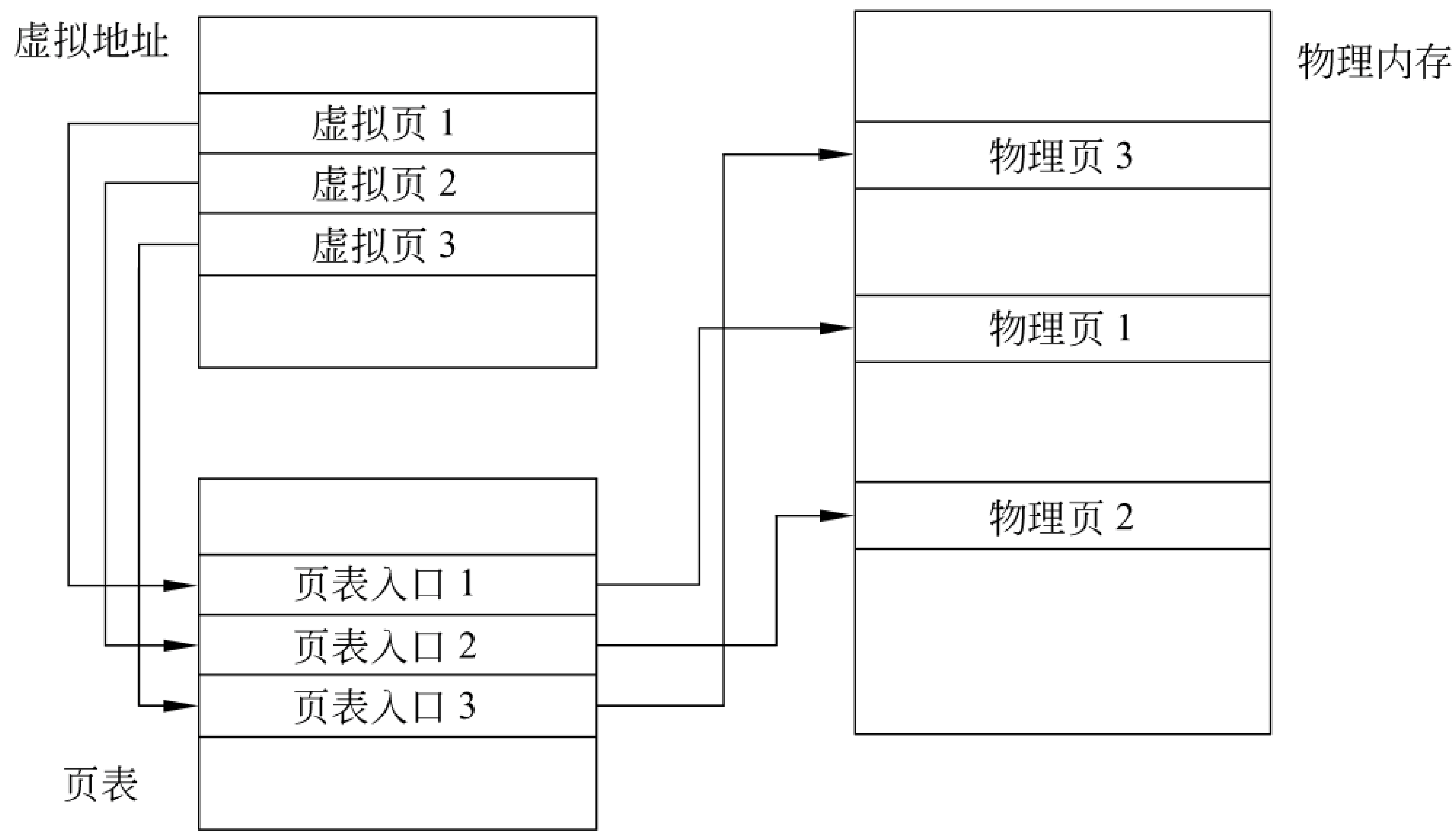



图 7.9 虚拟地址转换

为了将虚拟地址转换成以页为单位的结构，Windows 将一个 32 位的虚拟地址解释为 3 个独立的部分：页目录索引(Page Directory Index)、页表索引(Page Table Index)和字节索引(Byte Index)。如图 7.10 所示，页目录索引用 10 个地址位表示，用于定位虚拟地址所对应的页表；页表索引用 10 个地址位表示，用于定位该虚拟地址对应的页表入口；字节索引用 12 个地址位表示，用于确定该地址在对应的物理页上的具体位置(因为页面大小为 4K，刚好用 12 个地址位表示)。

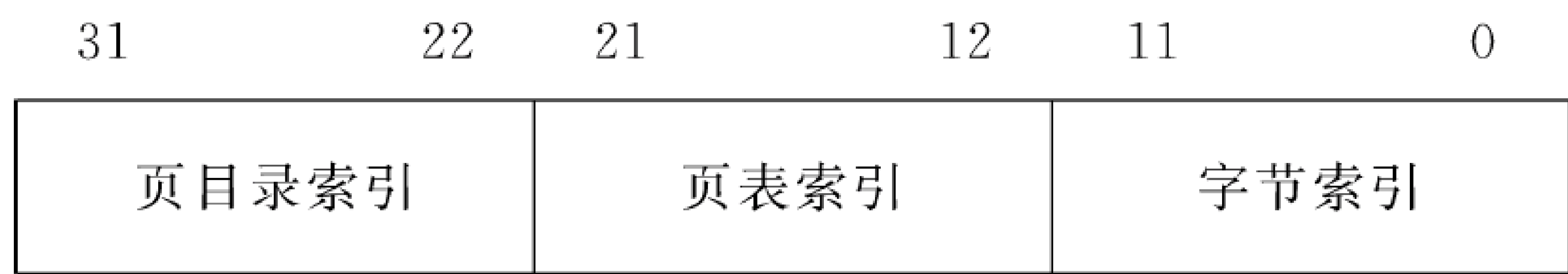



图 7.10 虚拟地址的页索引结构

我们用图 7.11 来说明虚拟地址通过页目录和页表进行地址转换的过程。

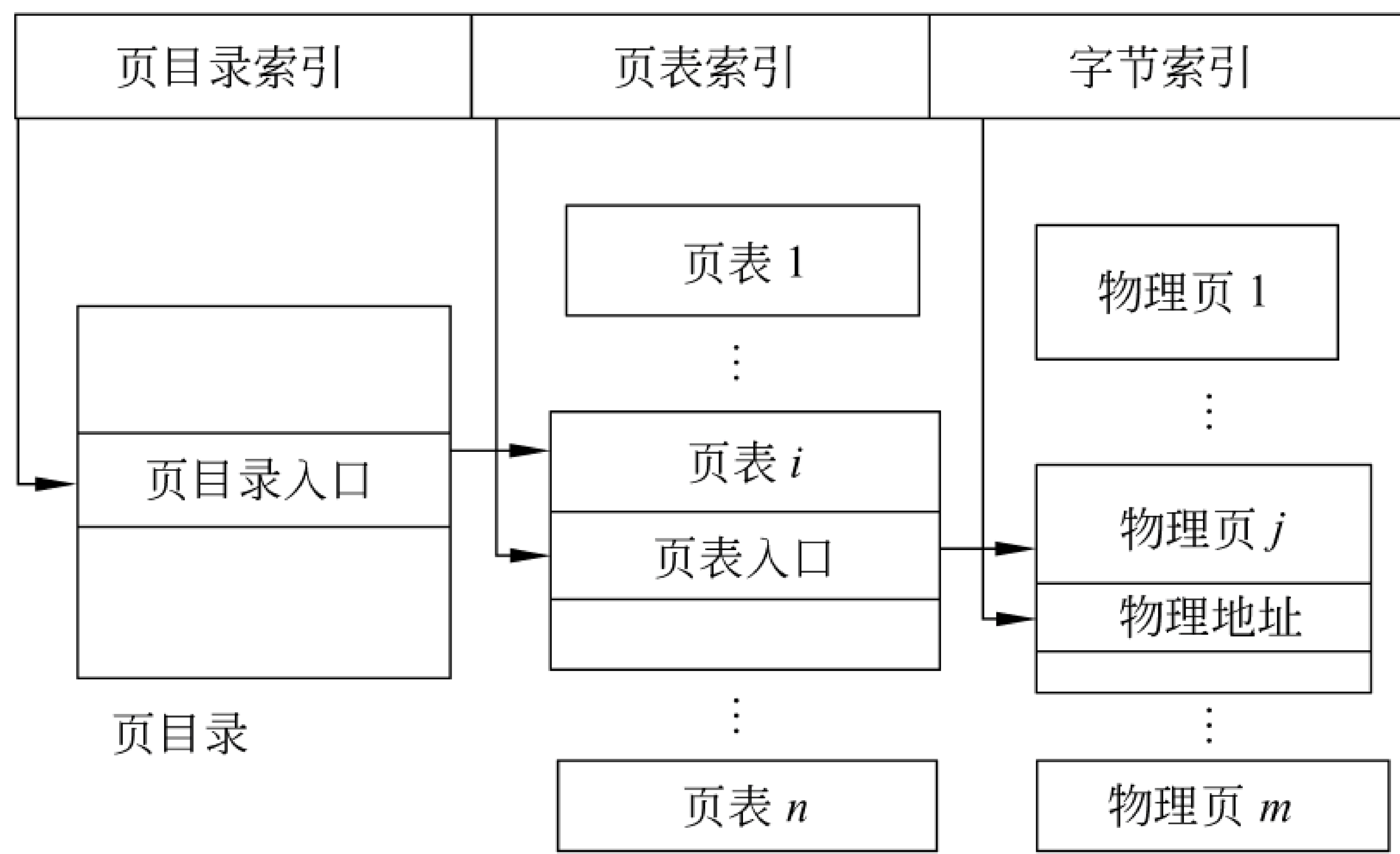



图 7.11 虚拟地址转换过程

第一步：每一个进程都对应一个页目录，当操作系统开始执行某一进程时，系统会设置当前进程所对应的页目录。

第二步：一个进程可以有多个页表，通过页目录索引，内存管理器可以定位相应的虚拟地址所对应的页表。

第三步：通过页表和页表索引，内存管理器可以定位虚拟地址对应的物理页框号。

第四步：当定位了物理页框号后，通过字节索引可以正确地判断虚拟地址对应的物理地址。

每一个进程都有单一页目录，它将该进程所有的页表都映射到一个页上，在 x86 体系结构的 Windows 中它有固定的系统虚拟地址：0xC030,0000。页表由一组页表入口构成，x86 系统的页表索引用 10 个地址位表示，因此一个页表可以索引 1024 个页表入口。对 32 位的 Windows 来说，需要 1024 个页表来映射 4GB 的虚拟地址空间。

如图 7.12 所示，一个页表入口不仅包含了虚拟地址所对应的物理页框号，而且包含了表示该内存页状态的标志，这些标志的含义在表 7.5 中列出。

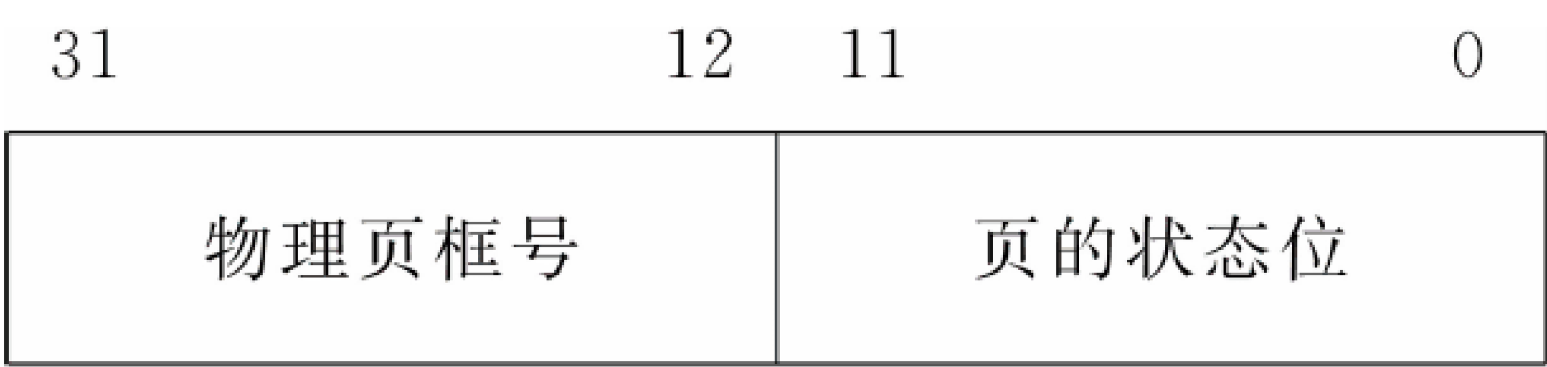



图 7.12　页表入口的结构

表 7.5 描述了页表入口的低 12 位表示的页的主要状态位及其含义。

**表 7.5　页表中的状态位及其含义**

| 状态位 | 含　　义 | 状态位 | 含　　义 |
|---|---|---|---|
| Accessed | 该页是否已被读过 | Owner | 该页是否可以在用户态下访问 |
| Cached Disabled | 该页是否不能被缓存 | Valid | 该页是否驻留在物理内存中 |
| Dirty | 该页是否已被写过 | Write Through | 是否跳过些缓存将该页实时写入磁盘 |
| Global | 该转换是否适用于所有的进程 | Write | 该页是否可写 |
| Large Page | 该页是否为大页(4MB) | | |

# 7.6　页 面 调 度

大多数计算机系统的实际可用物理内存都比运行应用所需的内存要小，在物理内存不够的情况下，内存管理器需要将一部分内存中的数据通过页面调度机制置换到磁盘上，以便为需要运行的数据和代码让出空间。当线程访问某一虚拟地址的数据，而它已经被置换到磁盘上时，虚拟内存管理器就会将这一部分内容重新从磁盘上调回到内存中。

## 7.6.1　缺页处理

当虚拟地址对应的页表入口的标志表明当前页不能马上被访问时，系统会判断原因并

发出缺页错误请求。系统会通过陷阱机制将这一错误转到内存管理器的错误处理程序。表 7.6 说明了各种缺页错误处理的方式。

表 7.6 缺页处理的各种情况

| 缺页原因 | 处理方式 |
| --- | --- |
| 被访问的页不在物理内存中,而在磁盘上的页文件或映射文件中 | 分配一个物理页,从磁盘上读入该页,并将它记入工作集 |
| 该页被挂起或在修改页列表中 | 将该页转移到进程或系统的工作集 |
| 访问预留页或超出了分配的地址空间 | 访问错误 |
| 在用户态访问只能在核心态访问的页 | 访问错误 |
| 写入只读的页 | 访问错误 |
| 写入"复制后写入"页 | 复制该页并将新的页作为进程的私有页,同时更新相应的映射和工作集 |
| 执行页中标明不能被执行的代码 | 访问错误 |

### 7.6.2 工作集及页面调度策略

工作集是驻留在物理内存中的虚拟页的集合,工作集分为进程工作集和系统工作集。进程工作集用来描述一个进程中的所有线程引用的驻留在内存中的页面,系统工作集表示系统空间中可被分页的系统代码和数据驻留在物理内存中的部分。

Windows 的页面调度采用的是请求式簇调度策略,即当线程产生缺页错误时,内存管理器如果判断该页不在物理内存中,它会将被请求的页面及和它相邻的一些页面调入内存。一般来讲,按照"集中访问",程序,特别是大型的程序,在特定的时间里仅在局部的地址空间上运行。根据这一原理调入缺页相邻的一些页,为线程随后访问它们做好了准备,大大减少了线程频繁进行缺页调度的几率。

当缺页处理需要调入新的页,而物理内存全部被占用时,Windows 采用"最久未使用"策略来决定哪些虚拟页需要从内存中移出去。Windows 为每个进程的工作集设置了页数限制,进程初始工作集大小为 50 个页面,可以通过系统调用来增加,但最大的工作集不超过 345 个页面。当物理内存不够且该进程的某些页面驻留时间最长,或该进程的工作集达到了最大限制时,系统就通过换页机制将这些页置换出物理内存。

### 7.6.3 页框号和物理内存管理

Windows 将物理内存按照页的大小顺序划分成同样大小的单元,称之为页框(page frame),系统用页框号数据库来描述整个物理内存的状态。

页框号数据库如图 7.13 所示,它描述了每一个物理内存页的状态,所有在工作集中的页面都标上了"激活"状态。为了让内存管理器快速地查找到同一状态的物理页,页框号数据库将相同状态的物理页通过链表的形式连接在一起。

表 7.7 对物理页的各种状态进行了说明。

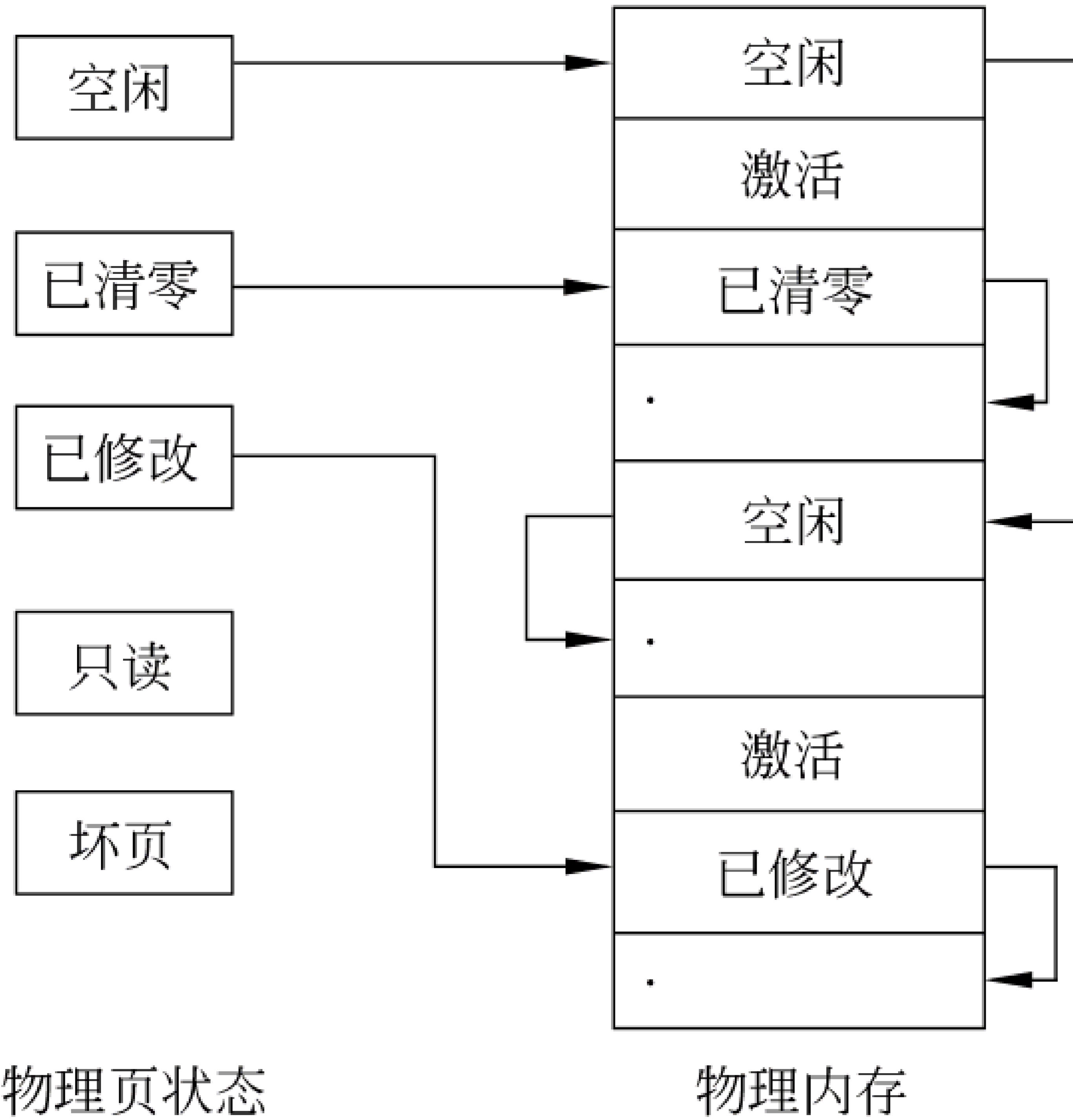



图 7.13 页框号数据库及状态链表

**表 7.7 物理页的状态及其说明**

| 状态 | 说　　明 |
|---|---|
| 激活 | 该页属于某一工作集，或者属于系统的常驻内存页 |
| 过渡 | 这是一个临时状态标志，表明该页既不属于一个工作集也不在分页列表上，而是正在被处理的过程中 |
| 挂起 | 该页被从一个工作集中移出了，对它的修改已经写入到磁盘，页表入口还是指向该物理页，但对访问设置了挂起标志 |
| 已修改 | 该页被从一个工作集中移出了，但对它的修改还没有被写入到磁盘，页表入口还是指向该物理页，完成写入磁盘后才可以投入使用 |
| 空闲 | 该页没有任何被修改的数据，但还没有被清零，出于安全性的考虑还不能被用户进程使用 |
| 已清零 | 该页已经被清零，能被用户进程使用 |
| 只读 | 该页位于只读内存 |
| 坏页 | 该页物理损坏，不能被使用 |

## 本章小结

Windows 操作系统有很多版本，其中 Windows NT 代表了 Windows 体系结构发展的主要方向。本章在介绍 Windows NT 体系结构的特点和相关概念后，着重介绍了 Windows 的进程管理和内存管理机制。

Windows 将处理器的执行模式分为核心态和用户态，操作系统的内核服务运行于核心态，而应用程序运行于用户态。用户态的进程有自己独立的地址空间，而核心态的服务和驱动程序共享同一系统地址空间。

本章详细地介绍了 Windows 进程和线程的关系。Windows 的所有系统服务和应用程序都是以进程的形式驻留在内存中，一个进程可以有一个或多个线程。进程为线程的运行提供资源和上下文环境，而线程通过处理器调度来完成计算的功能。Windows 以执行进程

块的形式驻留在内存中，而线程是以执行线程块的形式驻留在内存中的，它们之间相互关联。本章还介绍了执行进程块和执行线程块的数据结构。通过分析进程和线程的创建过程，本章介绍了 Windows 是如何管理进程和线程的。

本章还介绍了 Windows 的处理器调度机制，Windows 是以线程为调度对象的。每个线程都有调度优先级的设定，并且被分配了一定的执行时间片。调度器通过改变现成的状态来实现对多个线程的调度，Windows 采用基于优先级的抢占式调度算法。

本章介绍了 Windows 的内存管理机制，Windows 通过内存管理器来管理内存。32 位的 Windows 设计了大小为 4GB 的虚拟地址空间。其中低端的一半虚拟地址空间被分配作为进程的私有空间，高端的一半虚拟地址空间被分配给操作系统内核作为系统地址空间。

本章还介绍了系统内存和应用内存的不同分配方式，系统服务从统一的系统内存池内分配内存，而应用程序通过应用进程的页面管理和堆管理来分配内存。系统服务和应用程序是通过虚拟地址来操作内存的，访问之前需要将虚拟地址映射到实际的物理内存地址，内存管理器利用页表来进行地址转换。Windows 采用请求式簇调度策略来进行页面调度，并将驻留在物理内存中的页记入工作集。

## 习　　题

7.1　简述 Windows 核心态和用户态的区别。

7.2　Windows 操作系统有哪些系统服务？简述它们的功能。

7.3　描述 Windows 的进程和线程的概念，解释它们的区别和联系。

7.4　在 Windows 进程结构中，核心进程块和进程环境块分别起到什么作用？

7.5　简述 Windows 的线程结构以及它在内存中的驻留机制。

7.6　通过 Windows 任务管理器观察和分析系统中的进程。

7.7　用 C 语言编写程序利用 CreateProcess 和 CreateThread 函数创建一个 Windows 进程和两个线程。

7.8　哪些系统服务参与了 Windows 创建进程的过程？它们分别起到什么作用？

7.9　在 Windows 处理器调度的过程中，线程的哪些状态可以转换到就绪状态？它们在什么条件下转换到该状态？

7.10　简述 32 位的 Windows 操作系统的虚拟地址空间的布局。

7.11　简述虚拟地址到物理内存地址的转换过程，其中页表起到什么作用？

7.12　简述工作集在 Windows 内存管理中的作用和工作过程。

# 第 8 章 文件系统

无论是用户数据，还是计算机系统程序和应用程序，都要以一定的形式和格式进行组织、保存和管理。如何安全有效地快速大量处理这些数据和程序，就成为操作系统的重要内容。文件系统是计算机组织、存取和保存信息的重要手段。本章主要讨论文件的组织结构、存取结构、保护以及文件系统空间管理等问题。

## 8.1 文件系统的概念

### 1. 文件系统的引入

操作系统对计算机的管理包括两个方面：硬件资源的管理和软件资源的管理。硬件资源的管理包括 CPU 的管理、存储器的管理和设备管理等，主要解决硬件资源的有效和合理利用问题。软件资源的管理则包括对各种系统程序（包括操作系统本身的程序）、工具软件（例如编辑程序、编译程序等）、中间件、库函数及各种用户程序和数据的管理。图 8.1 给出了资源管理的分类图。

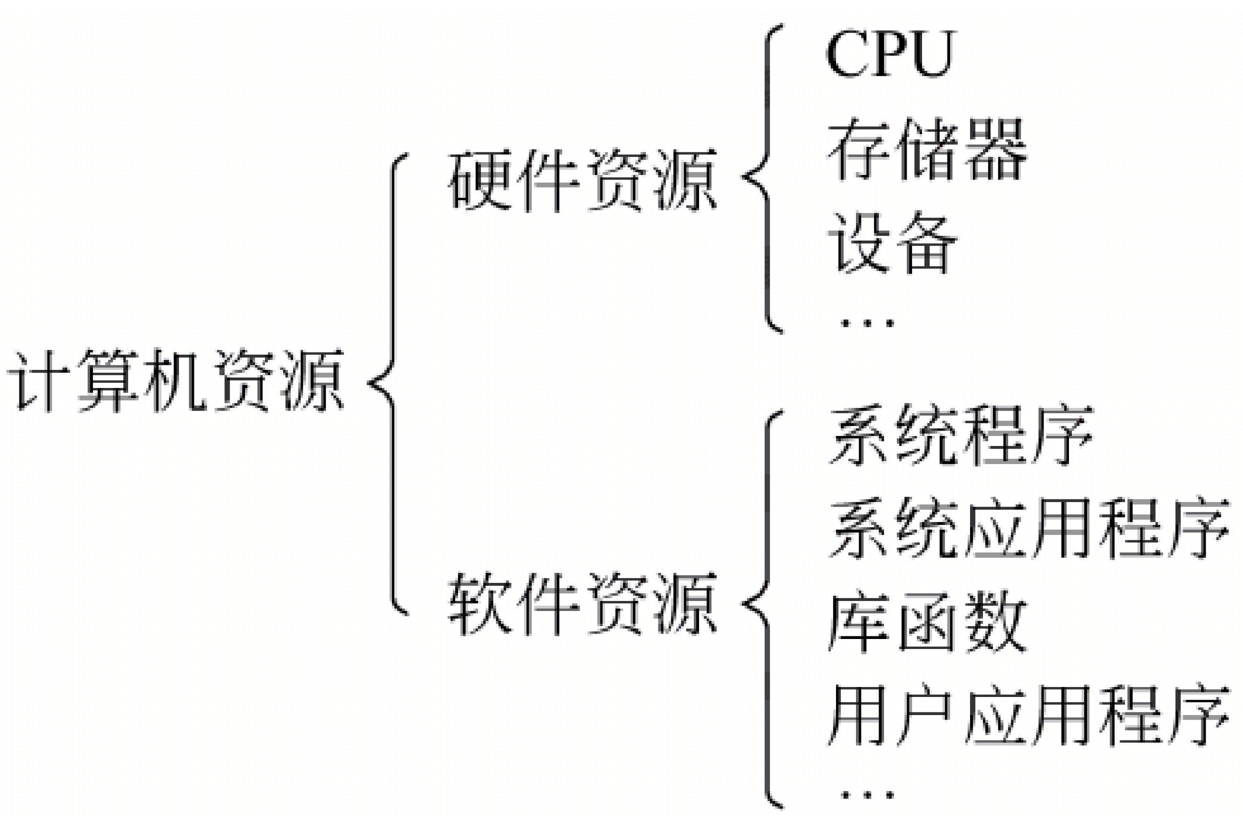


图 8.1 操作系统的软硬件管理

显然，用户使用计算机来完成自己的某件任务时，要遇到下列问题：

（1）使用现有的软件资源来协助完成自己的任务。例如，利用编辑、编译及链接程序来生成目标代码；利用系统调用库函数与实用程序来减少编程工作，避开与硬件有关的部分等。

（2）编制完成的或未完成的程序存放在什么地方，需要访问的数据存放在什么地方，从而使得人们可以再利用已有的软件资源。

事实上，这两个问题是一个怎样对软件资源（程序和数据）进行透明地快速存取问题。在早期的计算机系统中，由于硬件资源的限制，只能用卡片或纸带来存放程序或数据。这些卡片和纸带都分别编号存放，当用户需要使用它们时，再把这些卡片和纸带放在读卡机上输入计算机。显然，这些人工干预的控制和保存软件资源的方法不可能做到透明存取，不仅限制了计算机的处理能力和 CPU 等计算机硬件的利用率，而且不方便用户。

大容量直接存取的磁盘存储器以及顺序存取的磁带存储器等的出现，为程序和数据等软件资源的透明存取提供了物质基础。这导致了对软件资源管理质的飞跃——文件系统的出现。文件系统把程序和数据看作文件，并把它们存放在磁带或磁盘等大容量存储介质上，从而做到对程序和数据的透明存取。这里，透明存取是指不必了解文件存放的物理结构和查找方法等与存取介质有关的部分，只需给定一个代表某段程序或数据的文件名，文件系统就会自动地完成对与给定文件名相对应文件的有关操作。

文件系统必须完成下列工作：

（1）为了合理地存放文件，必须对磁盘等辅助存储器空间（或称文件空间）进行统一管理。在用户创建新文件时为其分配空闲区，而在用户删除或修改某个文件时，回收和调整存储区。

（2）为了实现按名存取，需要有一个用户可见的文件逻辑结构，用户按照文件逻辑结构所给定的方式进行信息的存取和加工。这种逻辑结构是独立于物理存储设备的。

（3）为了便于存放和加工信息，文件在存储设备上应按一定的顺序存放。这种存放方式被称为文件的物理结构。

（4）完成对存放在存储设备上的文件信息的查找。

（5）完成文件的共享和提供保护功能。

本章的后面各节将分别讨论这些问题。

**2. 文件与文件系统的概念**

1）文件

上面已经说过，文件是一段程序或数据的集合。这是一种较为模糊的说法。在计算机系统中，文件被解释为一组赋名的相关联字符流的集合，或者是相关联记录（一个有意义的信息单位）的集合。

文件的两种解释定义了两种文件形式。赋名的字符流文件是一种无结构文件或流式文件。目前常用的操作系统，例如 Windows 和 Linux 等均采用无结构文件形式。无结构文件由于采用字符流方式，与源程序和目标代码等在形式上是一致的，因此，该方式适用于源程序和目标代码等文件。由相关联记录组成的文件中的有些基本信息单位是记录。记录是由 $N(N>1)$ 个字节组成的具有特定意义的信息单位。记录式文件主要用于信息管理，例如数据库系统等。

在有些操作系统中，从字符流文件的角度出发，设备也被看作是赋予特殊文件名的文件。从而，系统可以对设备和文件实施统一管理，大大简化了设备管理程序和文件系统的接口设计。

用户文件名由用户给定，它是一个字母数字串，有些系统规定必须是英文字母开头且允许一些其他的符号出现在文件名的非开头部分。

2）文件系统

操作系统中与管理文件有关的软件和数据称为文件系统。它负责为用户建立、撤销、读写、修改和复制文件，还负责完成对文件的按名存取和进行存取控制。

文件系统具有以下特点：

（1）友好的用户接口，用户只对文件进行操作，而不管文件结构和存放的物理位置。

（2）对文件按名存取，对用户透明。

（3）某些文件可以被多个用户或进程所共享。

（4）文件系统大都使用磁盘、磁带和光盘等大容量存储器作为存储介质，因此，可存储大量信息。

**3. 文件的分类**

在文件系统中，为了有效、方便地管理文件，常常把文件按其性质和用途等进行分类。

按文件的性质和用途可以分为 3 类：

1）系统文件

该类文件只允许用户通过系统调用来执行它们，而不允许对其进行读写和修改。这些

文件主要由操作系统核心和各种系统应用程序和数据所组成。

2）库文件

该类文件允许用户对其进行读取和执行，但不允许对其进行修改。库文件主要由各种标准子程序库组成。如 C 语言子程序库、FORTRAN 子程序库等。

3）用户文件

用户文件是用户委托文件系统保存的文件。这类文件只由文件的所有者或所有者授权的用户才能使用。用户文件主要由源程序、目标程序和用户数据库等组成。

另外，按组织形式，文件又可被划分为以下 3 类：

1）普通文件

普通文件既包括系统文件，也包括用户文件、库函数文件和实用程序文件。普通文件主要是指组织格式为系统中所规定的最一般格式的文件，例如由字符流组成的文件。

2）目录文件

目录文件是由文件的目录信息构成的特殊文件。即该文件的内容不是各种程序或应用数据，而是用来检索普通文件的目录信息。

3）特殊文件

在 UNIX 系统中，所有的输入输出设备都被看作特殊文件。这组特殊文件在使用形式上与普通文件相同，如查找目录、存取操作等。但是，特殊文件的使用是和设备处理程序紧密相连的。系统必须把对特殊文件的操作转为对不同的设备的操作。

除了按文件的用途和组织形式来分类外，还可以按文件中的信息流向或文件的保护级别等分类。例如，按信息流向可把文件分为输入文件、输出文件以及输入/输出文件等。按文件的保护级别又可分为只读文件、读写文件、可执行文件和不保护文件等。

文件的分类主要是便于系统对不同的文件进行不同的管理，从而提高处理速度和起到保护与共享的作用。例如，一个系统文件在读入内存时将被放在内存的某一固定区且享受高的保护级别，从而不必像一般的用户文件那样只有在内存用户可用区分得相应的空闲区之后才能被调入内存。

## 8.2　文件的逻辑结构与存取方法

### 8.2.1　逻辑结构

文件的逻辑结构是用户可见结构。文件的逻辑结构可分为两大类：字符流式的无结构文件和记录式的有结构文件。在设计文件系统时，选择何种逻辑结构才能更有利于用户对文件信息的操作呢？一般情况下，选取文件的逻辑结构应遵循下述原则：

（1）当用户对文件信息进行修改操作时，给定的逻辑结构应能尽量减少对已存储好的文件信息的变动。

（2）当用户需要对文件信息进行操作时，给定的逻辑结构应使文件系统在尽可能短的时间内查找到需要查找的记录或基本信息单位。

（3）应使文件信息占据最小的存储空间。

（4）应便于用户进行操作。

显然，对于字符流的无结构文件来说，查找文件中的基本信息单位，例如某个单词，是比较困难的。但反过来，字符流的无结构文件管理简单，用户可以方便地对其进行操作。所以，那些对基本信息单位操作不多的文件较适于采用字符流的无结构方式，例如源程序文件、目标代码文件等。

除了字符流的无结构方式外，记录式的有结构文件可把文件中的记录按各种不同的方式排列，构成不同的逻辑结构，以便用户对文件中的记录进行修改、追加、查找和管理等操作。

记录是一个具有特定意义的信息单位，它由该记录在文件中的逻辑地址（相对位置）与记录名所对应的一组关键字、属性及其属性值所组成。图 8.2 是一个记录的组成示例。

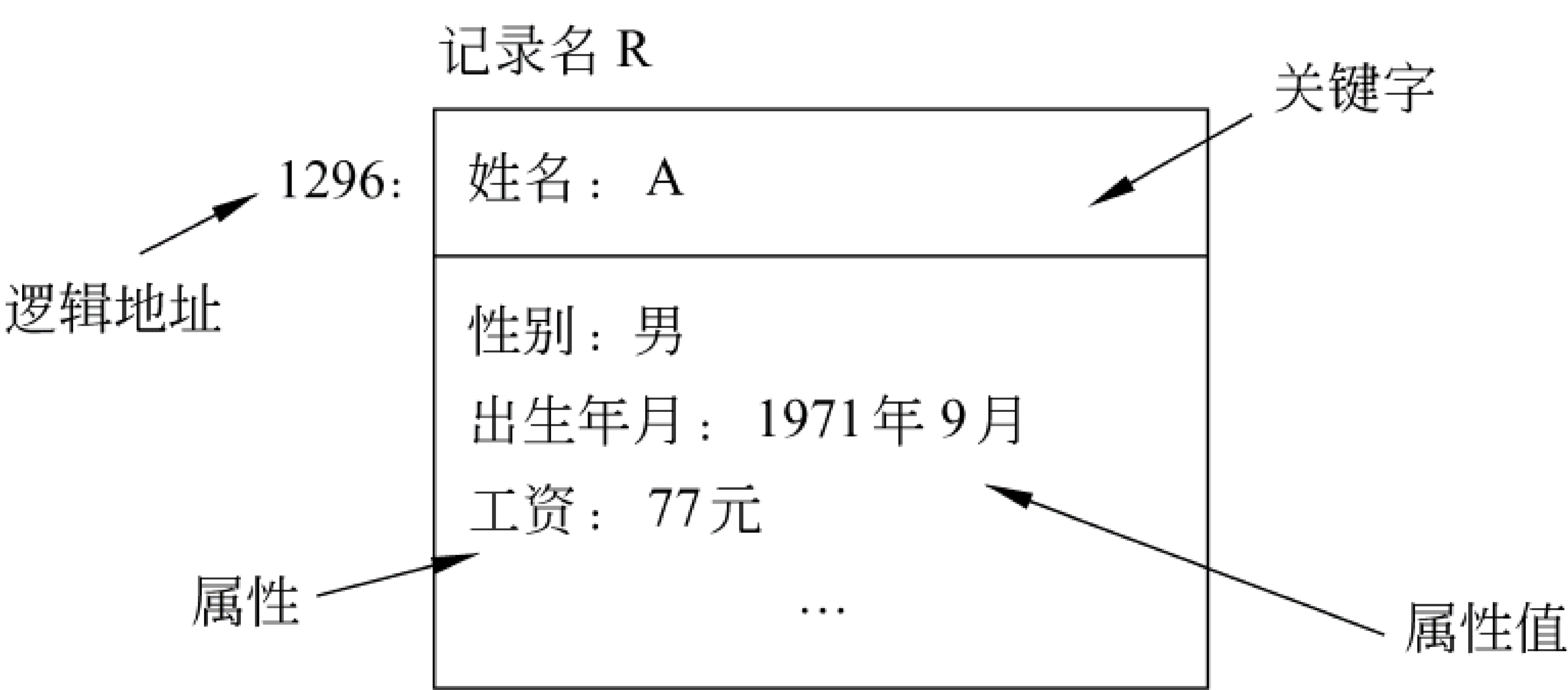



图 8.2　记录组成示例

图中，1296 是名为 R 的记录在文件中的逻辑地址，“姓名：A”是该记录的关键字，而“性别”、“出生年月”、“工资”等是该记录的属性，紧跟在这些后面的是属性值。一个记录可以有多个关键字，每个关键字可对应于多项属性。再者，根据各系统设计的要求不一样，记录既可以是定长的，也可以是变长的。记录的长度可以短到一个字符，也可以长到一个文件，这要由系统设计人员确定。

常用的记录式结构文件有连续结构、多重结构、转置结构和顺序结构。

下面分别介绍这几种结构。

**1. 连续结构**

连续结构是一种把记录按生成的先后顺序连续排列的逻辑结构。连续结构的特点是适用性强，可用于所有文件（字符流式的无结构文件实质上是记录长度为一个字符的连续结构文件），且记录的排列顺序与记录的内容无关。这有利于记录的追加与变更。但是，连续结构文件的搜索性能较差。例如，要找出某个指定关键字的记录时，系统必须对文件全体进行搜索。

**2. 多重结构**

如果把记录按关键字和记录名排列成行列式结构，则一个包含 $n$ 个记录名、$m(m\leqslant n)$ 个关键字的文件构成一个 $m*n$ 维行列式（见图 8.3）。其中，如果第 $i(1\leqslant i\leqslant m)$ 行和第 $j(1\leqslant j\leqslant n)$ 列所对应的位置上为 1，则表示关键字 $K_i$ 在记录 $R_j$ 中；反之，则表示关键字 $K_i$ 不在记录 $R_j$ 中。另外，同一个关键字也可以同时属于不同的记录。

$$
\begin{array}{c}
 \\ K_1 \\ K_2 \\ \vdots \\ K_m
\end{array}
\begin{array}{c}
\begin{array}{cccc} R_1 & R_2 & \cdots & R_n \end{array} \\
\begin{bmatrix}
1 & 0 & \cdots & 1 \\
0 & 0 & \cdots & 1 \\
\vdots & \vdots & \ddots & \vdots \\
1 & 1 & \cdots & 0
\end{bmatrix}
\end{array}
$$

图 8.3　文件的记录名和关键字构成的行列式

显然，如果只按行列式结构来排列记录，将会浪费较多的存储空间。从而，把行列式中那些为零的项去掉，并以关

键字 $K_i$ 为队首，以包含关键字 $K_i$ 的记录为队列元素来构成一个记录队列。对于一个有 $m$ 个关键字的队列来说，这样的队列有 $m$ 个。这 $m$ 个队列构成了该文件的多重结构(multi-list)，如图 8.4 所示。

**3. 转置结构**

在图 8.4 所示的多重结构中，每个队列中和关键字直接相连的只有一个记录。这种结构虽然在搜索时要优于连续结构，但在搜索某一特定记录时，必须在找到该记录所对应的关键字之后，再在该关键字所对应的队列中顺序查找。与此相反，转置结构(inverted file)把含有相同关键字的记录指针全部指向该关键字，也就是说，把所有与同一关键字对应的记录的指针连续地置于目录中该关键字的位置下(见图 8.5)。转置结构最适用于给定关键字后的记录搜索。

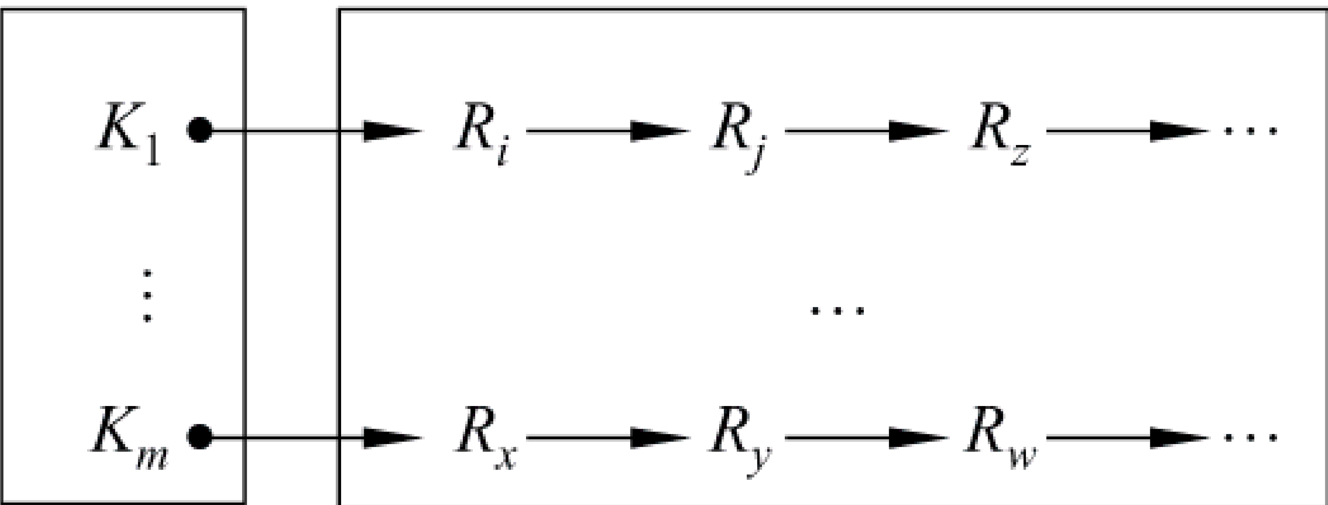


图 8.4 文件的多重结构

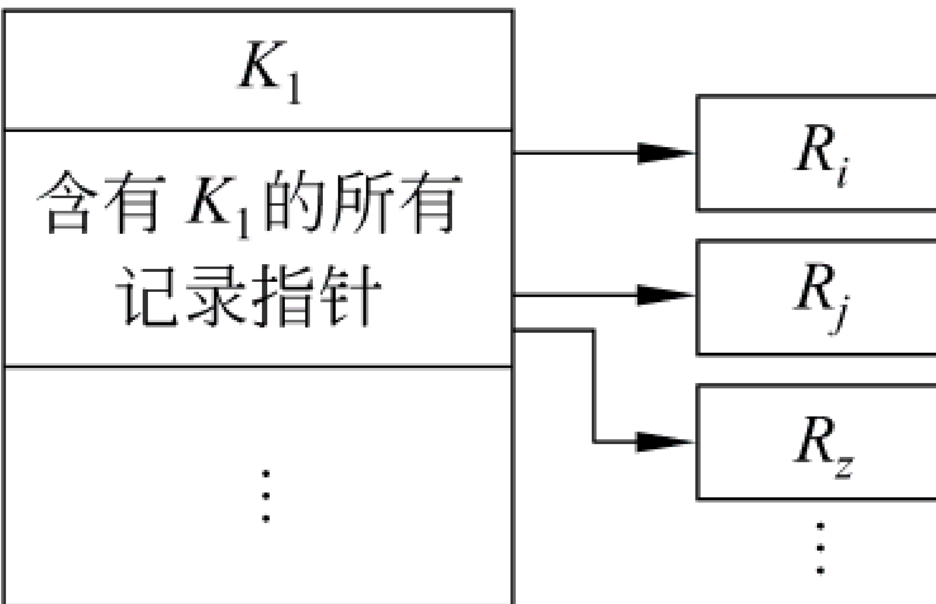


图 8.5 文件的转置结构

**4. 顺序结构**

如果系统要求按某种优先顺序来搜索或追加、删除记录，则最好采用顺序结构。如果给定了顺序规定(例如按字母顺序)，则把文件中的关键字按规定的顺序排列起来就形成了顺序结构文件。例如，把《人民日报》上登载的新闻按年月日为关键字做成记录放入文件中，并以时间先后顺序组成文件。这样，如果要处理某段时间内所发生的大事等问题就会变得非常简单。例如，用户想了解两伊战争的情况，则只要把 1990 年 8 月 19 日开始的两个月内的有关记录搜索到就行了。

## 8.2.2 存取方法

用户通过对文件的存取来完成对文件的修改、追加和搜索等操作。文件的存取是要找到文件内容所在的逻辑地址。常用的存取方法有 3 种：

(1) 顺序存取法；

(2) 随机存取法(直接存取法)；

(3) 按关键字存取法。

顺序存取是按照文件的逻辑地址顺序存取。在记录式文件中，这反映为按记录的排列顺序来存取，例如，若当前读取的记录为 $R_i$，则下一次读取的记录被自动地确定为 $R_i$ 的下一个相邻的记录 $R_{i+1}$。在无结构的字符流文件中，顺序存取反映当前读写指针的变化。在存取完一段信息之后，读写指针自动加上或减去该段信息长度，以便指出下一次存取时的位置。

随机存取法允许用户根据记录的编号来存取文件的任一记录，或者是根据存取命令把读写指针移到欲读写处来读写。

许多操作系统采用顺序存取和随机存取两种方法。

按关键字存取是一种用在复杂文件系统，特别是数据库管理系统中的存取方法。文件的存取是根据给定的关键字或记录名进行的。按关键字存取法首先搜索到要进行存取的记录的逻辑位置，再将其转换到相应的物理地址后进行存取。下面介绍按关键字存取的搜索方法。

对文件进行搜索的目的是要查找出特定记录所对应的逻辑地址，以便将其转换为相应的物理地址，实现对文件的操作。

对文件的搜索包括两种：关键字的搜索和记录的搜索。对关键字的搜索是在用户给定所要搜索的关键字和记录之后，确定该关键字在文件中的位置；而记录的搜索则是在搜索到所要查找的关键字之后，在含有该关键字的所有记录中查找出所需要的记录。显然，对于不同的逻辑结构的文件，其搜索方法和搜索效率都是不一样的。对指定记录 $R_i$ 的搜索过程如图 8.6 所示。

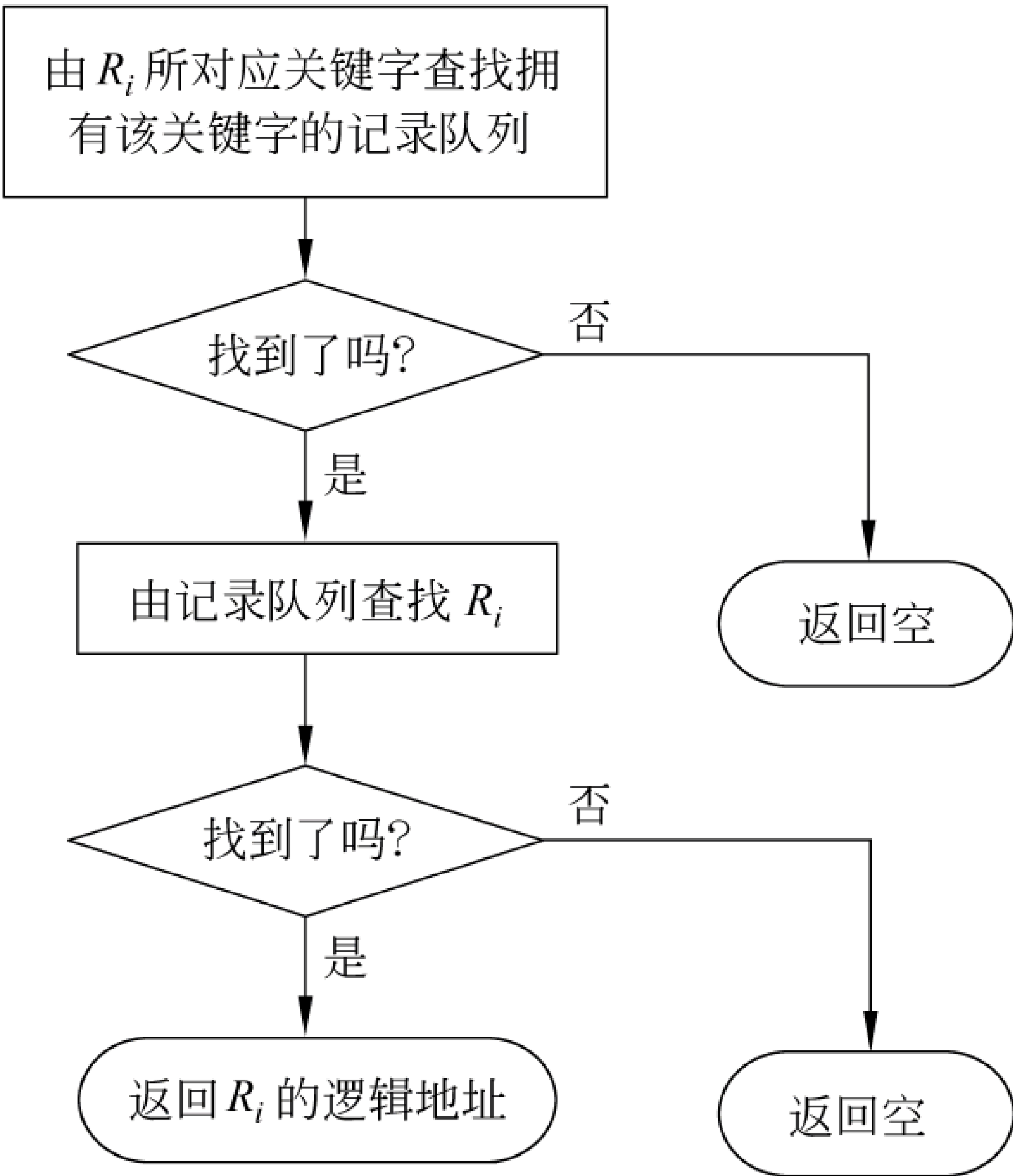



图 8.6　记录 $R_i$ 的搜索过程

对于给定的 $R_i$，首先，系统确定 $R_i$ 所对应关键字的记录队列。如果在所查找的文件中不存在这样的队列，则搜索算法结束返回，从而无法搜索到 $R_i$。如果找到 $R_i$，则返回其所对应的逻辑地址。如果找不到 $R_i$，则返回无法找到 $R_i$ 的有关信息。

对关键字或记录的搜索与其他数据搜索问题一样，都属于表格搜索问题(table lookup)。有许多搜索算法用来解决表格搜索问题。这些算法可以大致分为 3 种类型，即线性搜索法(linear search)、散列法(hash coding)和二分搜索法(binary search algorithm)。下面分别简单地介绍这几种搜索算法。

**1. 线性搜索法**

线性搜索法是一种最简单、最直观的搜索方法。它从第一个关键字或记录开始，依次和所要搜索的关键字或记录相比较，直到找到所需要的记录为止。线性搜索法所需要的搜索时间与所搜索的表格大小的 1/2 成正比。这是因为找到一个所需要的记录平均要和表中登记的总项数的 1/2 项比较后才能得到。

线性搜索法的搜索效率较低，在文件中记录个数较多时不宜采用。

**2. 散列法**

散列搜索法被广泛用于现代操作系统的数据查找。散列法的核心思想是：定义一个散列函数 $h(k)$，使得对于给定的关键字 $k$，散列函数 $h(k)$将其变换为 $k$ 所对应的逻辑地址。

在使用散列函数进行搜索时，有时会出现两个不同的输入值变换到同一地址的问题。即对于 $k_1 ! =k_2$，有 $h(k_1)=h(k_2)=A$。显然，$k_1$ 和 $k_2$ 中，至少有一个与 $A$ 中的内容不一致，也就是说，由散列变换得到的结果并不是所要搜索的关键字，这种问题称为散列冲突。解决散列冲突的方法是采用多次散列探索。例如，设第 $i$ 次散列变换的结果为 $h_i(k)$，$i=2,3,\cdots$，则可令

$$h_i(k) = (h_1(k) + d_i)(\bmod t)$$

这里，$t$ 为被搜索表格长度，$d_i$ 为第 $i$ 次搜索所得地址与第 1 次搜索所得地址之间的距离。$d_i$ 的取值方法很多，最简单的方法是设 $d_i$ 为 $i$ 的线性函数，即 $d_i = a * i$($a$ 为一个大于零的常数)。这种方法称线性散列法。但是，使用线性散列法并不能完全解决散列冲突问题。例如，对于 $i != j$，$k=1,2,\cdots$，如果 $h_i(k_1)=h_j(k_2)$，则存在 $h_i+k(k_1)=h_j+k(k_2)$。

解决散列冲突的另一个方法是生成一组随机数$\{r_1,r_2,\cdots,r_n\}$，且令 $d_i=r_i$。显然，除了 $h_i(k_i)=h_1(k_2)$可能存在之外，$h_i(k_i)=h_1+k(k_2)$的可能性很小，不过，使用随机数的方法需要占用一定的存储空间来生成和存放随机数组。

还有一个解决散列冲突的方法是采用平方散列函数方法，即令

$$h_i(k) = (h_1(k) + c * (i * i))(\bmod t)$$

这里，$t$ 是一个表示被搜索表格长度的素数，$c$ 是一个大于零的常数。可以证明，对于 $j>1(\bmod t)$，即使有 $h_1(k_1)=h_j(k_2)$，对于 $i=1,2,\cdots,t-1$，有 $h_1(k_1)! =h_j+i(k_2)$。

**3. 二分搜索法**

对于顺序结构排列的关键字或记录来说，二分搜索法具有较高的搜索效率。

设关键字 $K_0,K_1,K_2,\cdots,K_n(n>1)$按关键字间距 $d$ 排列，如果 $K_0$ 的逻辑位置为 $a_0$，则有 $K_i$ 的逻辑位置为 $a_0+i*d$。二分搜索法首先把所要搜索的关键字与队列的首尾关键字相比较，如果和其中之一相等，则返回所搜索到的关键字的逻辑位置。否则，再与队列 1/2 处的关键字比较，如果所要搜索的关键字正好等于该关键字的话，则返回该关键字的逻辑地址。否则，如果所要搜索的关键字 $K$ 小于位于队列中央的关键字的话，则继续搜索左边的半个队列；如果所要搜索的关键字 $K$ 大于位于队列中央的关键字的话，则继续搜索右边的半个队列。这样，每次用以中央关键字划分的部分组成新的队列反复进行上述搜索操作，直到找到为止。这一搜索过程如图 8.7 所示。

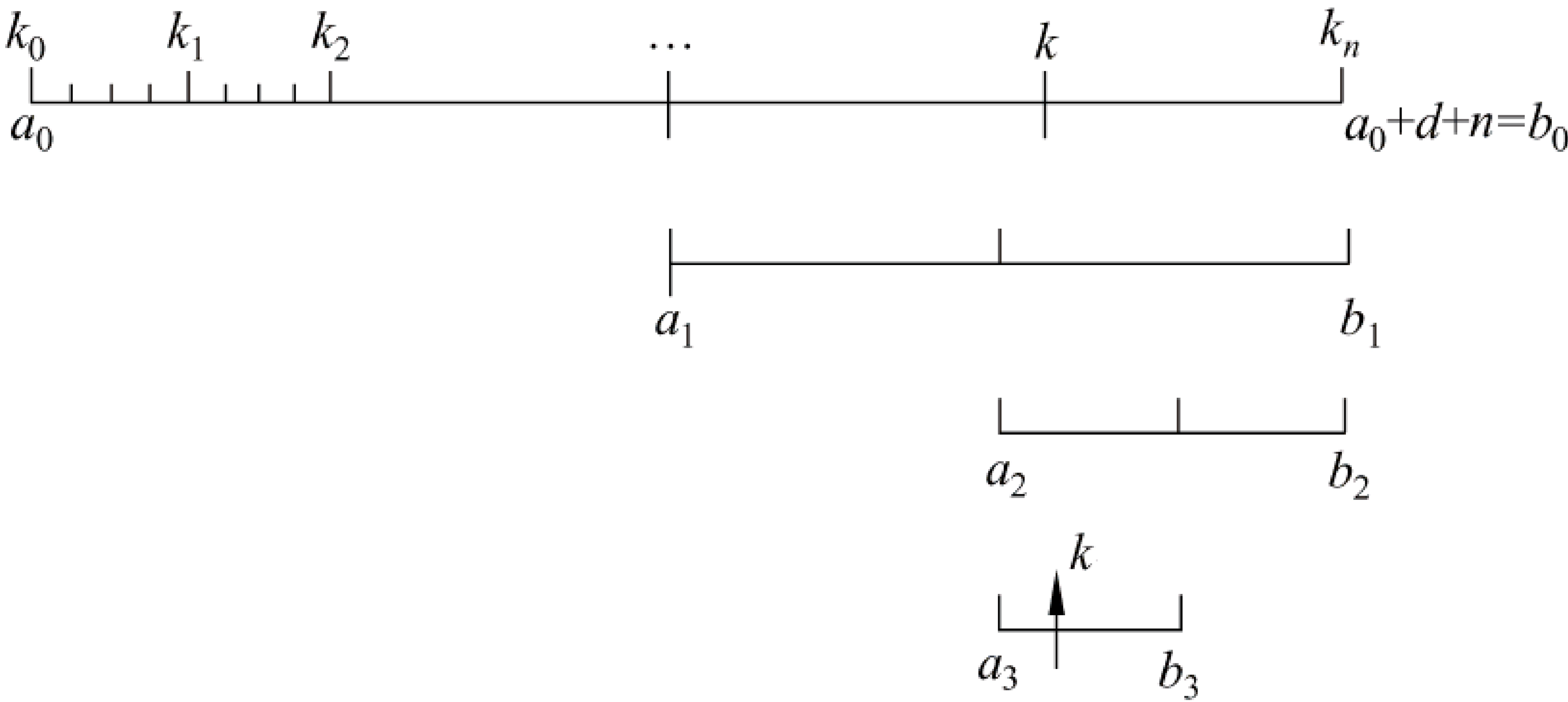



图 8.7　二分搜索法的搜索过程

二分搜索法的好处是搜索效率高。与线性搜索法相比，当 $n$(表长)=16 时，它比线性搜索法约快 2 倍；当 $n=1024$ 时，其平均搜索速度要快 50 倍。不过，二分搜索法需要事先把搜索对象按一定顺序排列。

## 8.3　文件的物理结构与存储设备

上节介绍了文件的逻辑结构和存取方法。用户对不同种类的文件采用不同的存取方法，以方便地对文件进行各种操作。无论是哪一种存取方法，都是首先搜索到操作对象——

记录或某段字符流信息的逻辑地址，然后，由逻辑地址映射到对应的物理地址，再对物理地址的有关信息进行操作。由逻辑地址到物理地址的映射是和文件的存储方式也就是文件的物理结构紧密相关的。另外，文件系统采用哪种存取方法和逻辑结构，实际上是和物理存储介质有关的。因此，本节介绍文件的物理结构，同时也介绍常用的文件存储设备。

## 8.3.1 文件的物理结构

在文件系统中，文件的存储设备通常划分为若干个大小相等的物理块，每块长为512B或1024B。与此相对应，为了有效地利用存储设备和便于系统管理，一般把文件信息也划分为与物理存储设备的物理块大小相等的逻辑块。从而，以块作为分配和传送信息的基本单位。显然，对于字符流的无结构文件来说，每一个物理块中存放长度相等的文件信息（存储文件尾部信息的物理块除外）。但是，对于记录式文件来说，由于记录长度既可以是固定的，也可以是可变的，而且其长度不一定刚好等于其物理块的长度，从而给由记录的逻辑地址到物理地址的变换带来了额外的负担。这里，为了简单起见，假设文件系统中每个记录的长度是固定的，且其长度正好等于物理块的长度。从而，对于记录式文件来说，利用上节讨论的搜索算法得到的逻辑地址正好与文件的逻辑块号一一对应，这就简化了所讨论问题的条件。

文件的物理结构是指文件在存储设备上的存放方法。事实上，由于文件的物理结构决定了文件信息在存储设备上的存储位置，因此，文件信息的逻辑块号（逻辑地址）到物理块号（物理地址）的变换也是由文件的物理结构决定的。

常用的文件物理结构如下。

**1. 连续文件**

连续文件是一种最简单的物理文件结构，它把一个在逻辑上连续的文件信息依次存放到物理块中。图8.8给出了连续文件结构的图形说明。在图8.8中，一个逻辑块号为0、1、2、3的文件依次存放在物理块10、11、12、13中。

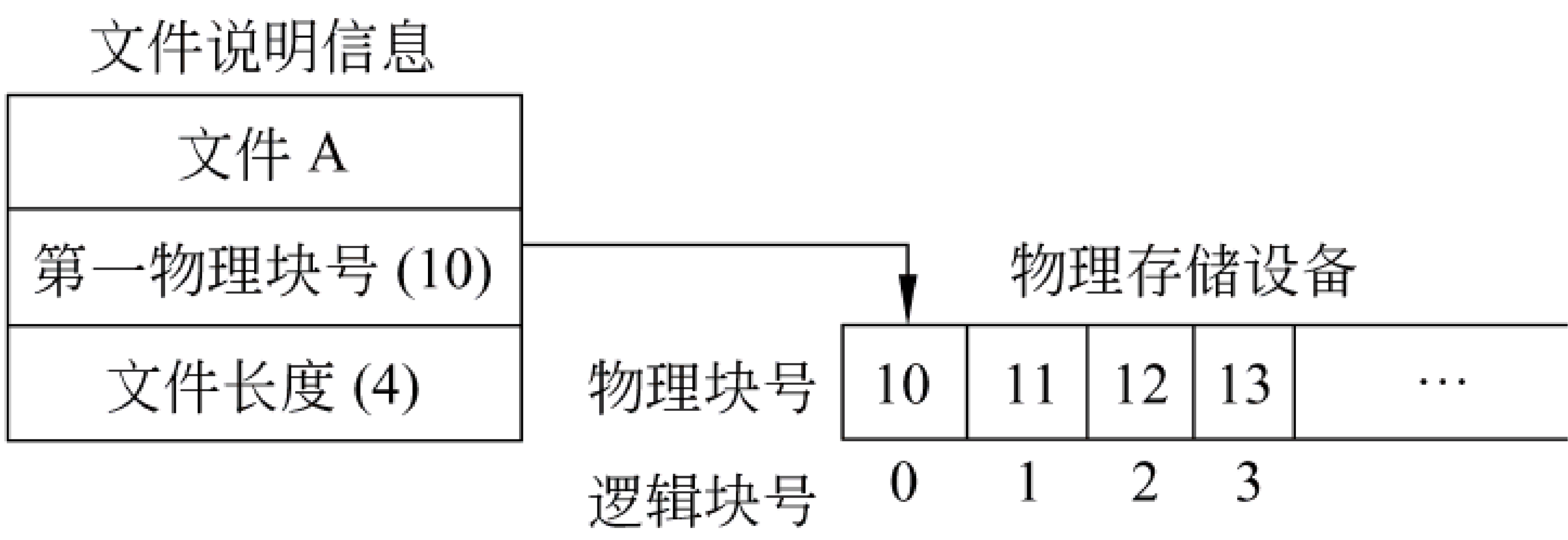



图8.8 连续文件结构

连续文件结构的优点是一旦知道了文件在文件存储设备上的起址和文件长度，就能很快地进行物理存取。这是因为文件的逻辑块号到物理块号的变换可以非常简单地完成。但是连续文件结构在建立文件时必须在文件说明信息中确定文件信息长度，且以后不能动态增长。而且在文件进行某些部分的删除后，又会留下无法使用的零头空间。因此，连续文件结构不宜用来存放用户文件、数据库文件等经常被修改的文件。

**2. 串联文件**

克服连续文件的缺点的办法之一是采用串联文件结构。串联文件结构用非连续的物理块来存放文件信息。这些非连续的物理块之间没有顺序关系，其中每个物理块设有一个指针，指向其后续连接的另一个物理块，从而使得存放同一文件的物理块链接成一个串联队

列。图 8.9 给出了串联文件的物理结构。

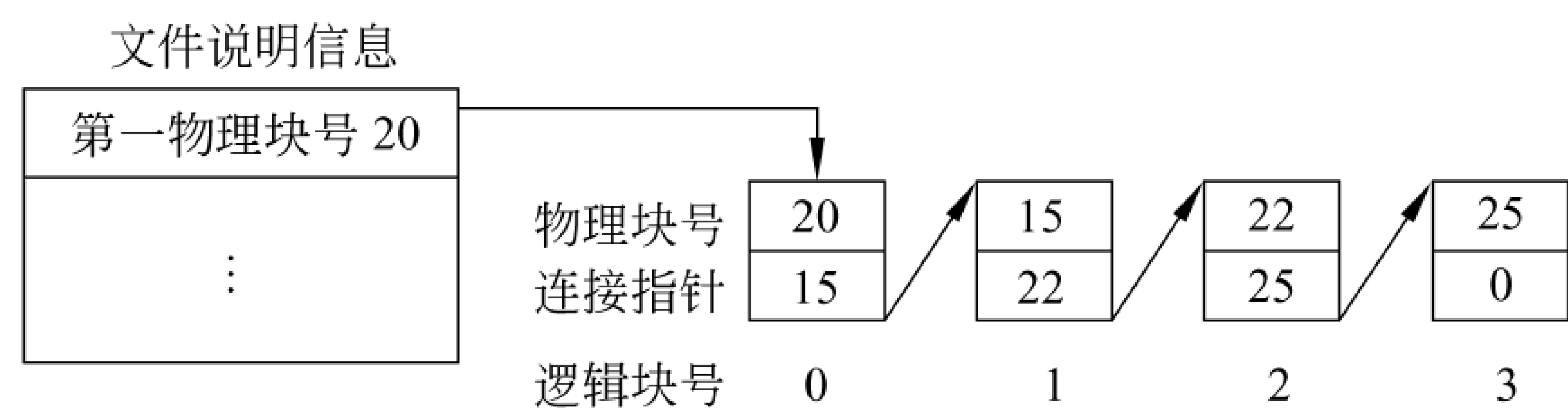



图 8.9　串联文件结构

显然，使用串联文件结构时，不必在文件说明信息中指明文件的长度，只需指明该文件的第一个块号就行了。串联文件结构的另一个特点是文件长度可以动态地增长，只要调整连接指针就可在任何一个信息块之间插入或删除一个信息块。

采用串联文件结构时，逻辑块到物理块的转换由系统沿串联队列查找与逻辑块号对应的物理块号的办法完成。例如，在图 8.9 所示的文件结构中，如果用户所要进行操作的逻辑块号为 2，则系统从第一个物理块 20 开始，一直沿串联队列搜索到队列中逻辑块号为 2 的第三块时，得到其所对应的物理块号为 22。

由于串联文件结构只能按队列中的串联指针顺序搜索，因此，串联文件结构的搜索效率较低。串联文件结构一般只适用于逻辑上连续的文件，且存取方法应该是顺序存取的。否则，为了读取某个信息块而造成的磁头大幅度移动将花去较多的时间。因此，串联文件结构不适宜随机存取。

**3. 索引文件**

第三种文件物理结构是索引结构。索引结构要求系统为每个文件建立一张索引表，表中指出文件信息所在的逻辑块号和与之对应的物理块号。索引表的物理地址则由文件说明信息项给出。索引文件结构如图 8.10 所示。

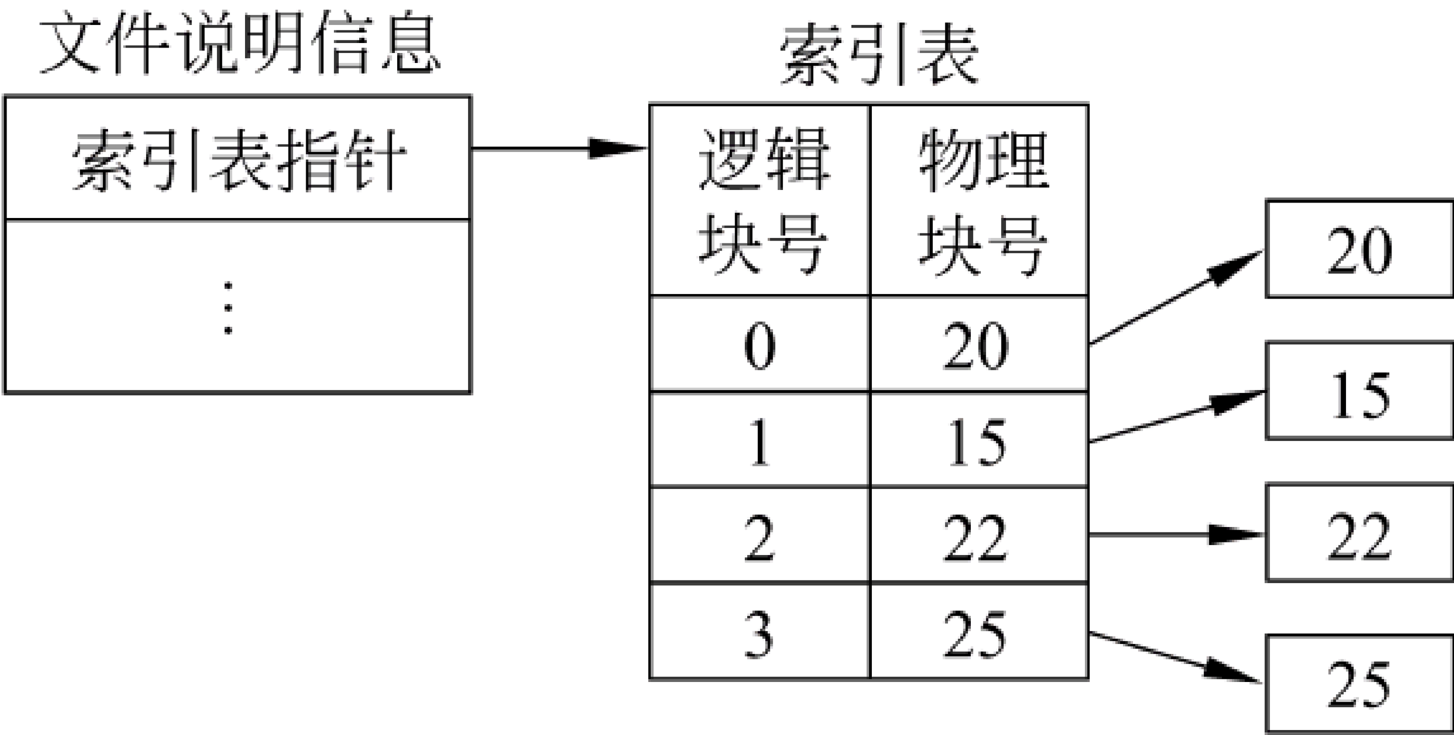



图 8.10　索引文件结构

索引文件结构既可以满足文件动态增长的要求，又可以较为方便和迅速地实现随机存取。因为有关逻辑块号和物理块号的信息全部放在一个集中的索引表中，而不是像串联文件结构那样分散在各个物理块中。

在很多情况下，有的文件很大，文件索引表也就较大。如果索引表的大小超过了一个物理块，那么我们必须像处理其他文件的存放那样决定索引表的物理存放方式，但这不利于索引表的动态增加；索引表也可按串联方式存放，但这却增加了存放索引表的时间开销。一种较好的解决办法是采用间接索引(多重索引)，也就是在索引表所指的物理块中存放的不是文件信息，而是装有这些信息的物理块地址。这样，如果一个物理块可装下 $n$ 个物理块地址的话，则经过一级间接索引，可寻址的文件长度将变为 $n*n$ 块。如果文件长度还大于 $n*n$ 块的话，还可以进行类似的扩充，即二级间接索引。其原理如图 8.11 所示。

不过，大多数文件不需要进行多重索引，也就是说，这些文件所占用的物理块号可以放在一个物理块内。如果对这些文件也采用多重索引，则显然会降低文件的存取速度。因此，在实际系统中，总是把索引表的头几项设计成直接寻址方式，也就是这几项所指的物理块中

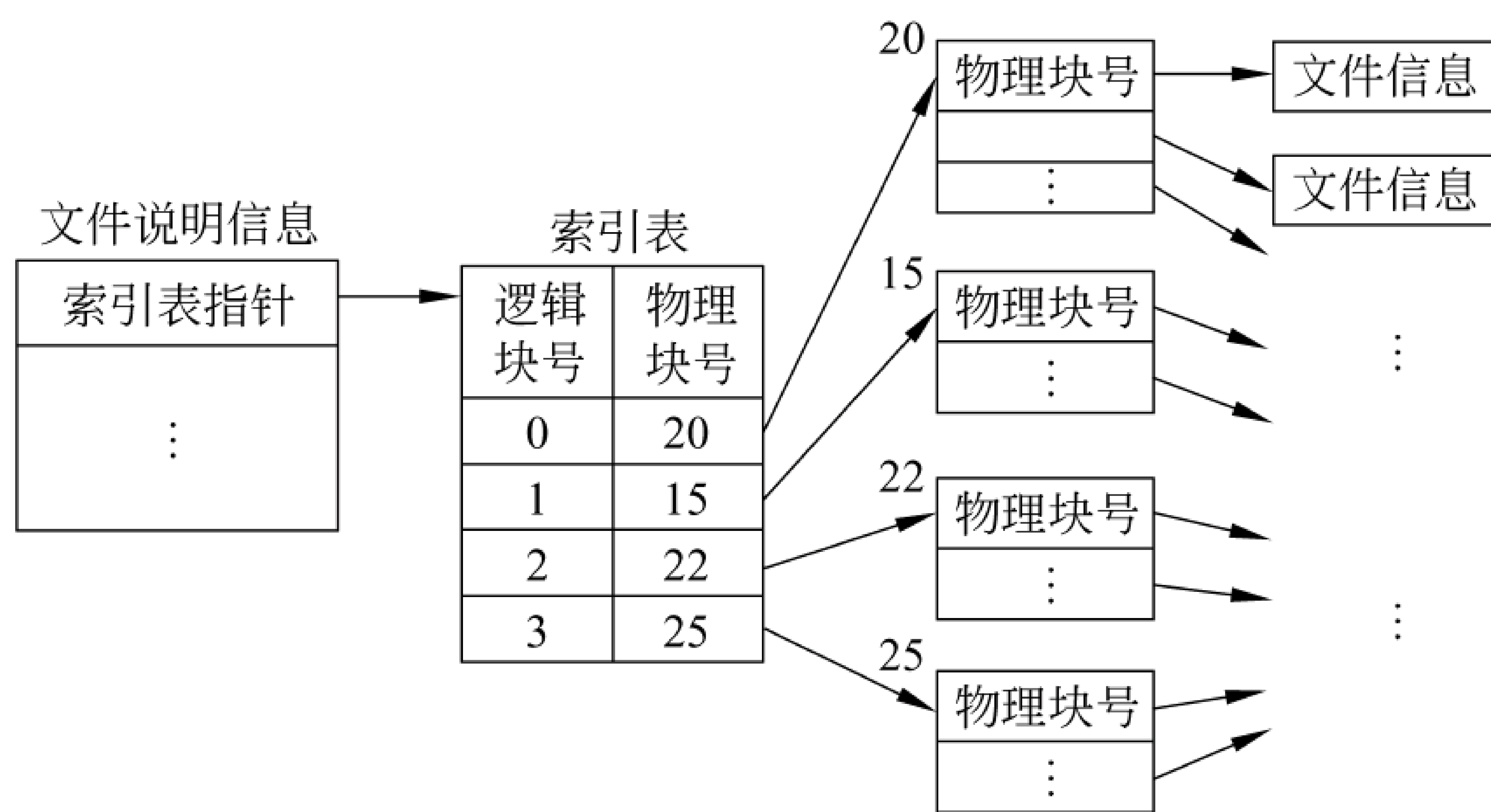



图 8.11　多重索引结构

存放的是文件信息；而索引表的后几项设计成多重索引，也就是间接寻址方式。在文件较短时，就可利用直接寻址方式找到物理块号而节省存取时间。

索引结构既适用于顺序存取，也适用于随机存取。索引结构的缺点是由于使用了索引表而增加了存储空间的开销。另外，在存取文件时需要至少访问存储器两次以上。其中，一次是访问索引表，另一次是根据索引表提供的物理块号访问文件信息。由于文件在存储设备的访问速度较慢，因此，如果把索引表放在存储设备上，势必大大降低文件的存取速度。一种改进的方法是，当对某个文件进行操作之前，系统预先把索引表放入内存。这样，文件的存取就可直接在内存通过索引表确定物理地址块号，而访问磁盘的动作只需要一次。

## 8.3.2　文件存储设备

存储设备有磁盘、光盘和磁带等。其中磁盘又可分为硬盘和软盘。近年来，软盘已逐步消失，取而代之的是光盘和优盘。由于存储设备的特性决定了文件的存取方法，因此，这里介绍以磁带为代表的顺序存取存储设备和以磁盘为代表的直接存取存储设备的特性及有关存取方法。

**1. 顺序存取存储设备**

磁带是一种最典型的顺序存取存储设备。顺序存取存储设备只有在前面的物理块被存取过之后，才能存取后续的物理块的内容。而且，为了在存取一个物理块时让磁带机提前加速和不停止在下一个物理块的位置上，磁带的两个相邻的物理块之间设计有一个间隙将它们隔开（见图 8.12）。




图 8.12　磁带的结构

磁带设备的存取速度或数据传输率与下列因素有关：

（1）信息密度（字符数/英寸）；

（2）磁带带速（英寸/秒）；

（3）块间间隙。

如果带速高，信息密度大，且所需块间隙（磁头启动和停止时间）小的话，则磁带存取速度和数据传输率高，反之则磁带存取速度和数据传输率低。

另外，由磁带的读写方式可知，只有当第 $i$ 块被存取之后，才能对第 $i+1$ 块进行存取操作。因此，某个特定记录或物理块的存取访问与该物理块到磁头当前位置的距离有很大关系。如果相距甚远，则要花费很长的存取时间来移动磁头。因此，如果按随机方式或按关键字存取方式存取磁带上的文件信息的话，其效率不会很高。但是，磁带设备具有容量大、顺序存取方式时存取速度高等优点。因此，磁带设备获得了较为广泛的应用。

**2. 直接存取存储设备**

磁盘是最典型的直接存取存储设备。磁盘设备允许文件系统直接存取磁盘上的任意物理块。为了存取一个特定的物理块，磁头将直接移动到所要求的位置上，而不需要像顺序存取那样事先存取其他的物理块。

磁盘机种类很多，但一般由一些磁盘片组成的磁盘组组成。其中每个磁盘片对应一个装有读/写磁头的磁头臂，磁头臂上的两个读/写磁头分别对磁盘片的上下两面进行读写。系统在对磁盘进行初始化处理时，把每个磁盘片分割成一些大小相等的扇区。在磁盘转动时经过读/写磁头所形成的圆形轨迹称为磁道。由于磁头臂可沿半径方向移动，因此，磁盘上有多条磁道。另外，人们通常把所有磁盘片的相同磁道称为一个柱面，因此，磁盘上每个物理块的位置可用柱面号、磁头号和扇区号表示，这些地址和物理块号一一对应。磁盘的结构如图 8.13 所示。

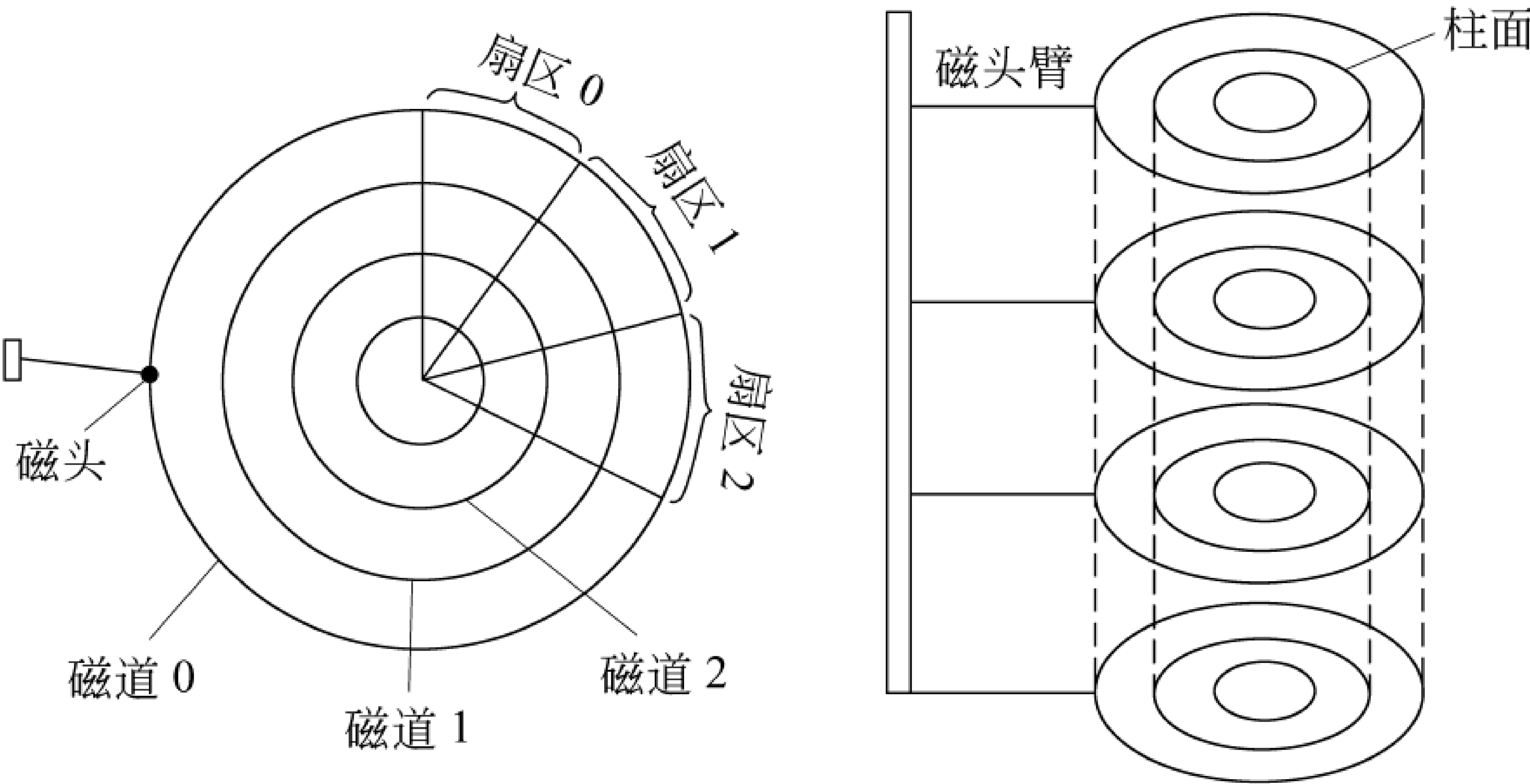



图 8.13　磁盘的结构

有关存储设备，特别是硬盘存储设备的进一步信息，读者可从网站 www.storage.com 上了解。

## 8.4　文件存储空间管理

存储空间管理是文件系统的重要任务之一。只有有效地进行存储空间管理，才能保证多个用户共享文件存储设备和得以实现文件的按名存取。由于文件存储设备是分成若干个大小相等的物理块，并以块为单位来交换信息的，因此，文件存储空间的管理实质上是一个空闲块的组织和管理问题，它包括空闲块的组织、空闲块的分配与空闲块的回收等几个问题。

有下述3种不同的空闲块管理方法：

（1）空闲文件目录；

（2）空闲块链；

（3）位示图。

下面介绍这几种管理方法。

**1. 空闲文件目录**

最简单的空闲块管理方法就是把文件存储设备中的空闲块的块号统一放在一个称为空闲文件目录的物理块中。其中空闲文件目录的每个表项对应一个由多个空闲块构成的空闲区，它包括空闲块个数、空闲块号和第一个空闲块号等。

在系统为某个文件分配空闲块时，首先扫描空闲文件目录项，如找到合适的空闲区项，则分配给申请者，并把该项从空白文件目录中去掉。如果一个空闲区项不能满足申请者要求，则把目录中另一项分配给申请者（连续文件结构除外）。如果一个空闲区项所含块数超过申请者的要求，则为申请者分配了所要的物理块之后，再修改该表项。

当一个文件被删除，释放存储物理块时，系统则把被释放的块号、长度以及第一块块号置入空白目录文件的新表项中。

在内存管理时讨论过有关空闲连续区分配和释放算法，只要稍加修改就可用于空闲文件项的分配和回收。

空闲文件项方法适用于连续文件结构的文件存储区的分配与回收。

**2. 空闲块链**

空闲块链是一种较常用的空闲块管理方法。空闲块链把文件存储设备上的所有空闲块链接在一起，当申请者需要空闲块时，分配程序从链头开始摘取所需要的空闲块，然后调整链首指针。反之，当回收空闲块时，把释放的空闲块逐个插入链尾上。

空闲块链的链接方法因系统而异，常用的链接方法有按空闲区大小顺序链接的方法、按释放先后顺序链接的方法以及成组链法。其中成组链法可被看作空闲块链的链接法的扩展。

按空闲区大小顺序链接和按释放先后顺序链接的空闲块管理，在增加或移动空闲块时需对空闲块链做较大的调整，因而有一定的系统开销。成组链法在空闲块的分配和回收方面则要优于上述两种链接法。下面介绍成组链法的基本原理。

成组链法首先把文件存储设备中的所有空闲块按50块划分为一组。组的划分为从后往前顺次划分（见图8.14）。其中，每组的第一块用来存放前一组中各块的块号和总块数。由于第一组的前面已无其他组存在，因此，第一组的块数为49块。不过，由于存储设备的空间块不一定正好是50的整倍数，因而最后一组将不足50块，且由于该组后面已无另外的空闲块组，所以，该组的物理块号与总块数只能放在管理文件存储设备用的文件资源表中。

在成组链法对文件设备进行了上述分组之后，系统可根据申请者的要求进行空闲块的分配，并在释放文件时回收空闲块。下面介绍成组链法的空闲块分配和释放过程。

首先，系统在初启时把文件资源表复制到内存，从而使文件资源表中放有最后一组空闲块块号与总块数的堆栈进入内存，并使得空闲块的分配与释放可在内存进行。这就减少了每次分配和释放空间都要启动I/O设备的压力。

与空闲块块号及总块数相对应，用于空闲块分配与回收的堆栈有栈指针$P_{tr}$，且$P_{tr}$的初

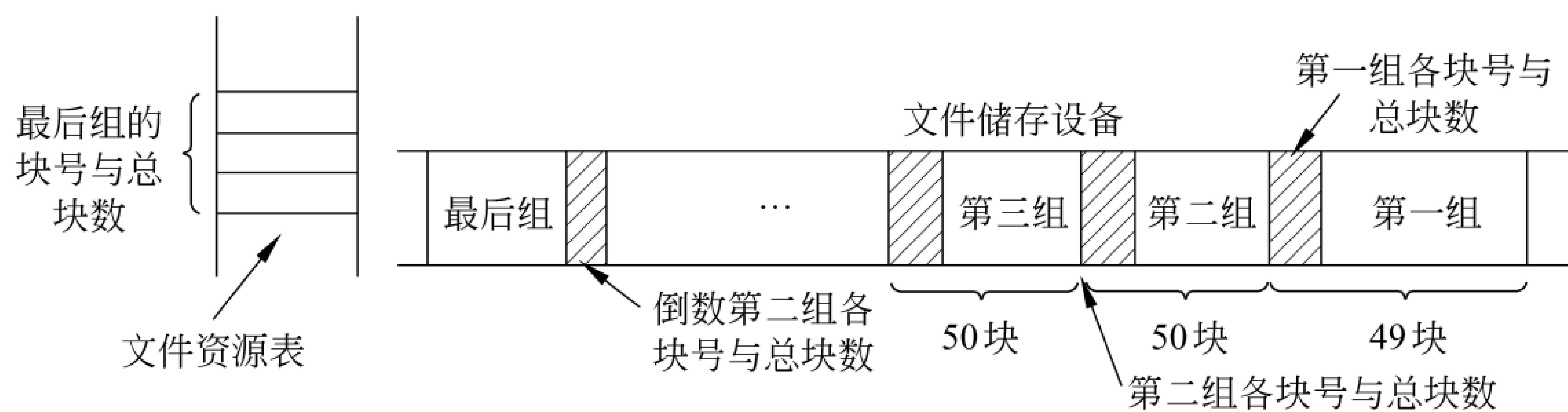



图 8.14　成组链法的组织

值等于该组空闲块的总块数。当申请者提出空闲块要求 $n$ 时，按照后进先出的原则，分配程序在取走 $P_{tr}$ 所指的块号之后，再做 $P_{tr} \leftarrow P_{tr} - 1$ 的操作。这个过程一直持续到所要求的 $n$ 块都已分配完毕或堆栈中只剩下最后一个空闲块的块号。当堆栈中只剩下最后一个空闲块号时，系统启动设备管理程序，将该块中存放的下一组的块号与总块数读入内存之后将该块分配给申请者。然后，系统重新设置 $P_{tr}$ 指针，并继续为申请者进程分配空闲块。

文件存储设备的最后一个空闲块中设置有尾部标识，以指示空闲块分配完毕。

如果用户进程不再使用有关文件并删除这些文件时，回收程序回收装有这些文件的物理块。成组链法的回收过程仍利用文件管理堆栈进行。在回收时，回收程序先做 $P_{tr} \leftarrow P_{tr} + 1$ 操作，然后把回收的物理块号放入当前指针 $P_{tr}$ 所指的位置。如果 $P_{tr}$ 等于 50，则表示该组已经回收结束。此时，如果还有新的物理块需要回收的话，回收该块并启动 I/O 设备管理程序，把回收的 50 个块号与块数写入新回收的块中。然后，将 $P_{tr}$ 重新置 1 另起一个新组。

显然，对空闲块的分配和释放必须互斥进行，否则将会发生数据混乱。

**3. 位示图**

空闲文件目录和空闲块链法在分配和回收空闲块时，都需在文件存储设备上查找空闲文件目录项或链接块号，这必须经过设备管理程序启动外设才能完成。为了提高空闲块的分配和回收速度，人们使用位示图的方法进行空闲块管理。

系统首先从内存中划出若干个字节，为每个文件存储设备建立一张位示图。这张位示图反映每个文件存储设备的使用情况。在位示图中，每个文件存储设备的物理块都对应一个比特位。如果该位为 0，则表示所对应的块是空闲块；反之，如果该位为 1，则表示所对应的块已被分配出去。

显然，利用位示图来进行空闲块分配时，只需查找图中的 0 位，并将其置为 1 位。反之，利用位示图回收时只需把相应的比特位由 1 改为 0 即可。

## 8.5　文件目录管理

为了实现对文件的按名存取，首先，每个文件必须有一个文件名与其对应。不同文件类型的文件名由不同的人员指定，一般来说，用户文件名由用户指定，系统文件和特殊文件在系统设计时指定。

为了有效地利用存储空间以及迅速准确地完成由文件名到文件物理块的转换，我们必须把文件名及其结构信息等按一定的组织结构排列，以方便文件的搜索。把文件名和对该

文件实施控制管理的控制管理信息称为该文件的文件说明,并把一个文件说明按一定的逻辑结构存放到物理存储块的一个表目中。利用文件说明信息,可以完成对文件的创建、检索以及维护作用。因此,把一个文件的文件说明信息称为该文件的目录。对文件目录的管理就是对文件说明信息的管理。

文件目录的管理除了要解决存储空间的有效利用之外,还要解决快速搜索、文件命名冲突以及文件共享问题。下面分别说明。

### 8.5.1　文件的组成

从文件管理角度看,一个文件包括两部分:文件说明和文件体。

文件体指文件本身的信息,它可能是前面各节讨论的记录式文件或字符流式文件。

文件说明有时也叫文件控制块(FCB),它至少包括文件名、与文件名相对应的文件内部标识以及文件信息在文件存储设备上第一个物理块的地址(物理结构是边连续结构时)。另外,根据系统要求不同,它还包括关于文件逻辑结构、物理结构、存取控制和管理等的信息等。这里的管理信息主要指访问时间和记账信息等。

文件说明组成目录文件。文件系统利用目录文件完成按名存取和对文件信息的共享与保护。

### 8.5.2　文件目录

文件目录可分为单级目录、二级目录和多级目录。

单级目录是一种最简单、最原始的目录结构。文件系统为存储设备的所有文件建立一张目录表,每个文件在其中占有一项用来存放文件说明信息。该目录表存放在存储设备的某固定区域,在系统初启时或需要时,系统将其调入内存(或部分调入内存)。文件系统通过该表提供的信息对文件进行创建、搜索和删除等操作。例如,当建立一个文件时,首先从该表中申请一项,并存入有关说明信息;当删除一个文件时,就从该表中删去一项。

严格地说,利用单级目录,文件系统就可实现对文件系统空间的自动管理和按名存取。例如,当用户进程要求对某个文件进行读写操作时,调用有关系统调用由事件驱动或中断总控方式进入文件系统,此时,CPU 控制权在文件系统手中。文件系统首先根据用户给定的文件名搜索单级文件目录表,以查找文件信息的物理块号。如果搜索不到对应的文件名,则失败返回(读操作时),或由空闲块分配程序进行空闲块分配后,再修改单级目录表。如果已找到对应的第一个物理块块号,则根据文件对应的物理结构信息计算出所要读写的信息块物理块块号,然后把 CPU 控制权交给设备管理系统启动设备进行读写操作。单级目录时的文件系统读写处理过程如图 8.15 所示。

不过,由于在单级目录表中,各文件说明项都处于平等地位,只能按连续结构或顺序结构存放,因此,文件名与文件必须一一对应。如果两个不同的文件重名的话,则系统将把它们视为同一文件。另外,由于在单级目录中必须对所有文件信息项进行搜索,因而,搜索效率也较低。

为了改变单级目录中文件命名冲突问题和提高对目录表的搜索速度,单级目录被扩充成二级目录。

在二级目录结构中,各个文件的说明信息被组织成目录文件,且以用户为单位把各自的

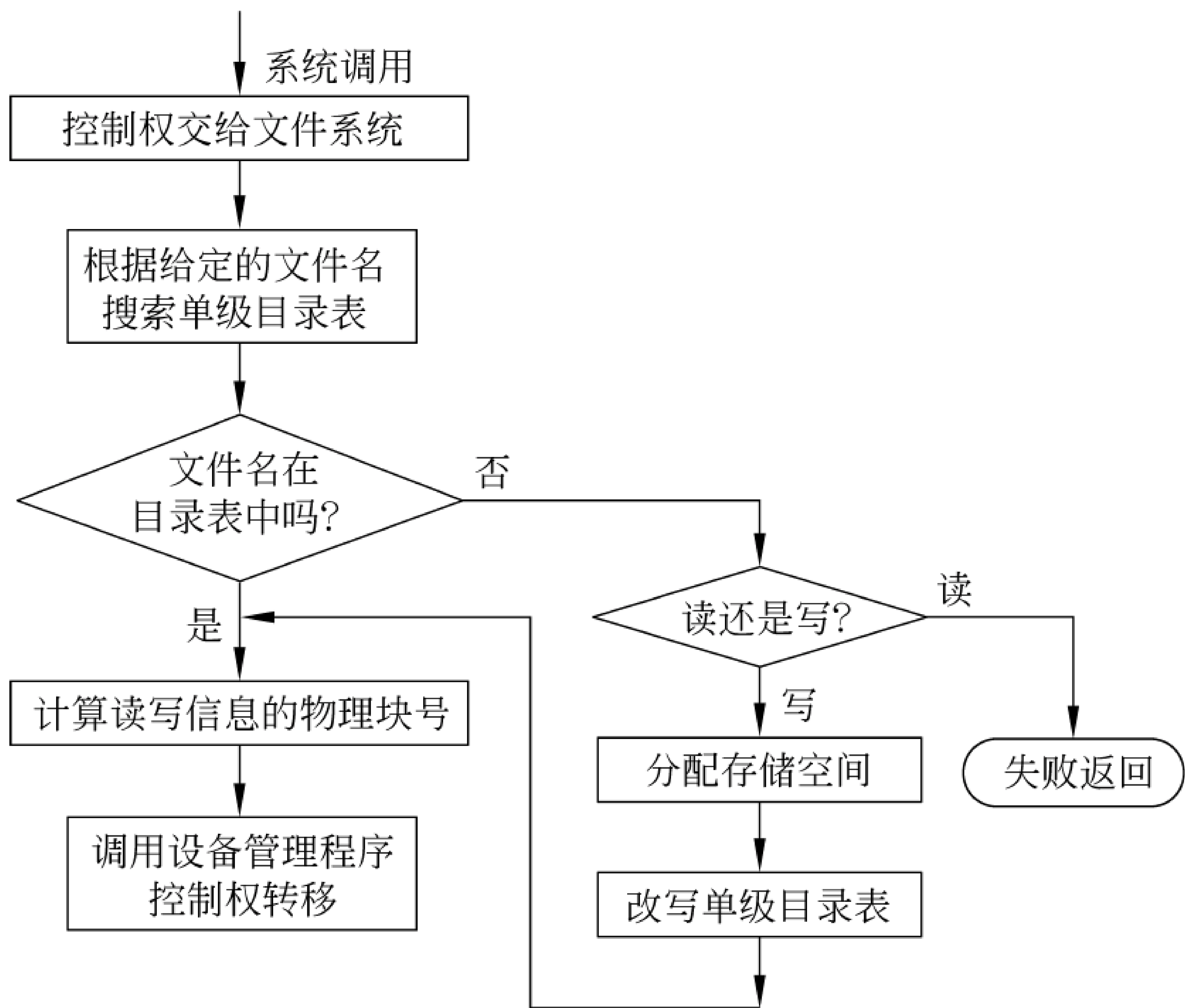



图 8.15　单级目录的读写处理过程

文件说明划分为不同的组。然后,这些不同的组名有关的存取控制信息存放在主目录(MFD)的目录项中。与 MFD 相对应,用户文件的文件说明所组成的目录文件被称为用户文件目录(UFD)。这样,由 MFD 和 UFD 就形成二级目录,其结构如图 8.16 所示。

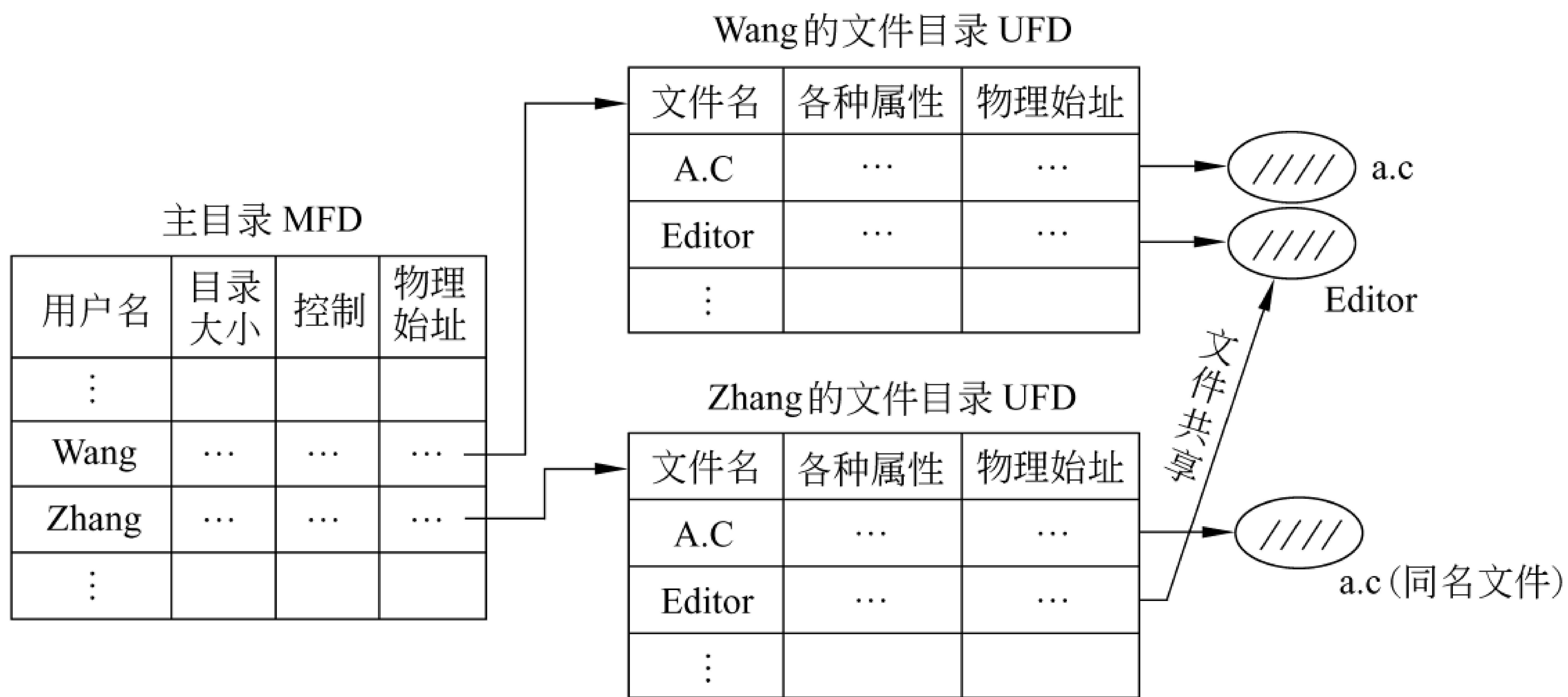



图 8.16　二级目录结构

当用户要对一个文件进行存取操作或创建、删除一个文件时,首先从 MFD 找到对应的目录名,并从用户名查找到该用户的 MFD。余下的操作与单级目录时相同。

使用二级目录可以解决文件重名和文件共享问题,并可获得较高的搜索速度。由于采用二级目录时首先从 MFD 开始搜索,因此,从系统管理的角度来看,文件名已演变成为用户名/用户文件名。从而,即使两个不同的用户具有同名文件,系统也会把它们区别开来。再者,利用二级目录,也可以方便地解决不同用户间的文件共享问题,这只要在被共享的文件说明信息中增加相应的共享管理项,并把共享文件的文件说明项指向被共享文件的文件说明项即可。

另外，与单级目录相比，如果单级目录表的长度为 $n$ 的话，则单级目录时的搜索时间与 $n$ 成正比；在二级目录时，由于 $n$ 的目录已被划分为 $m$ 个子集，则二级目录的搜索时间是与 $m+r$ 成正比的。这里的 $m$ 是用户个数，$r$ 是每个用户的文件的个数。一般有 $m+r\leqslant n$，从而二级目录的搜索时间要快于单级目录。

把二级目录的层次关系加以推广，就形成了多级目录。在多级目录结构中，除了最低一级的物理块中装有文件信息外，其他每一级目录中存放的都是下一级目录或文件的说明信息。由此形成层次关系，最高层为根目录，最低层为文件。多级目录构成树形结构，如图 8.17 所示。

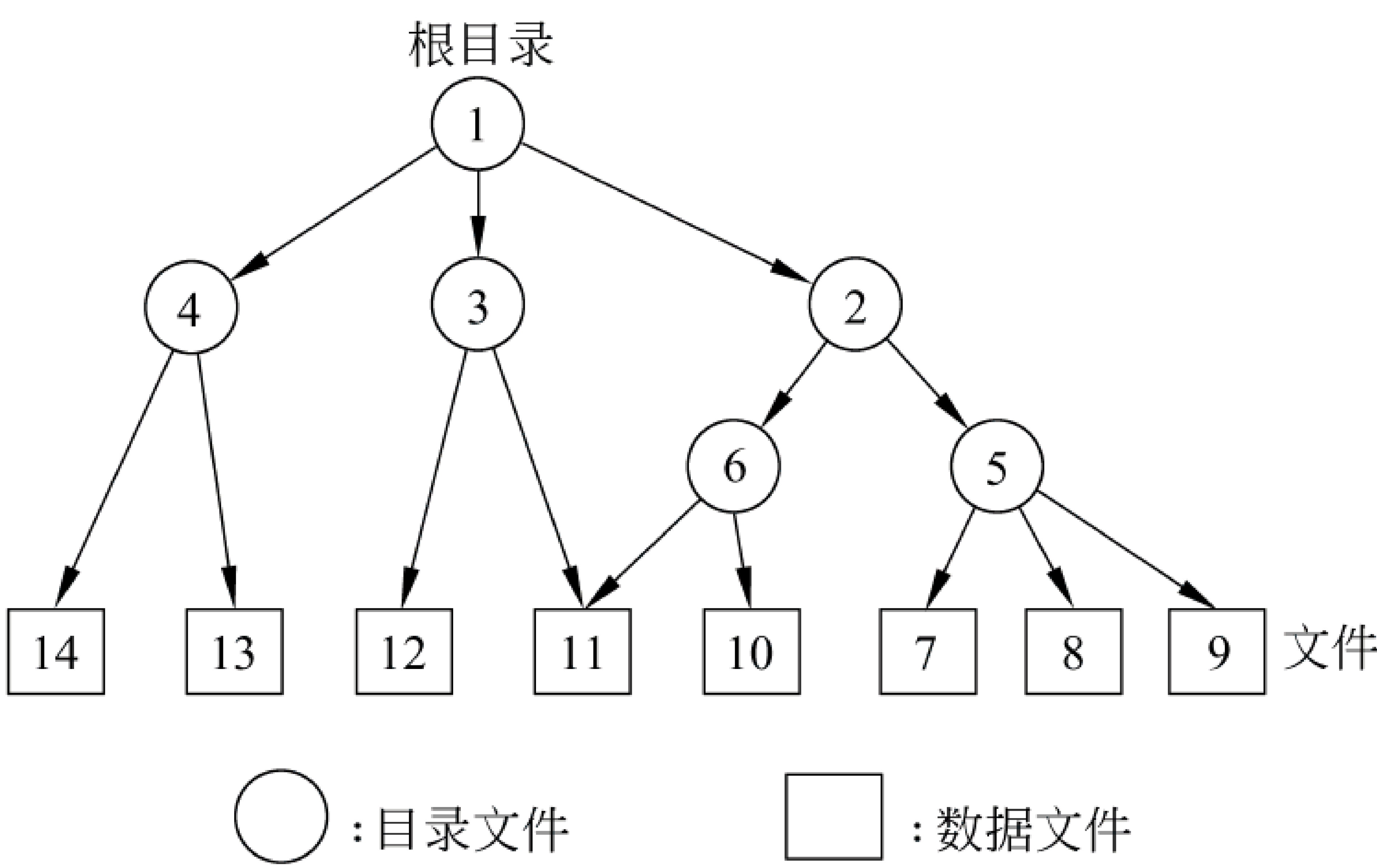



图 8.17　文件系统的树形结构

树形结构的多级目录具有下列特点：

（1）层次清楚。由于采用分支结构，不同性质、不同用户的文件可以构成不同的子树，便于管理。不同层次、不同用户的文件可以被赋予不同的存取权限，有利于文件的保护。

（2）解决了文件重名问题。文件在系统中的搜索路径是从根开始到文件名为止的各文件名组成的，因此，只要在同一子目录下的文件名不发生重复，就不会由文件重名而引起混乱。

（3）查找搜索速度快。在 8.2 节讨论的对文件中关键字的各种搜索方法，例如线性搜索法、散列法以及二分搜索法都可用来对各级目录进行查找。由于对多级目录的查找每次只查找目录的一个子集，因此，其搜索速度较单级目录和二级目录时更快。

### 8.5.3　便于共享的文件目录

文件系统的一个重要任务就是为用户提供共享文件信息的手段。这是因为对于某一个公用文件来说，如果每个用户都在文件系统内保留一份该文件的副本，这将极大地浪费存储空间。如果系统提供了共享文件信息的手段，则在文件存储设备上只需存储一个文件副本，共享该文件的用户以自己的文件名去访问该文件的副本就可以了。

从系统管理的观点看，有 3 种方法可以实现文件共享：

（1）绕道法；

（2）链接法；

（3）基本文件目录表（BFD）。

绕道法要求每个用户处在当前目录下工作，用户对所有文件的访问都是相对于当前目

录进行的。用户文件的固有名(为了访问某个文件而必须访问的各个目录和文件的目录名与文件名的顺序连接称为固有名)是由当前目录到信息文件通路上所有各级目录的目录名加上该信息文件的符号名组成。使用绕道法进行文件共享时,用户从当前目录出发向上返回到与所要共享文件所在路径的交叉点,再顺序下访到共享文件。绕道法需要用户指定所要共享文件的逻辑位置或到达被共享文件的路径。绕道法的原理如图 8.18 所示。

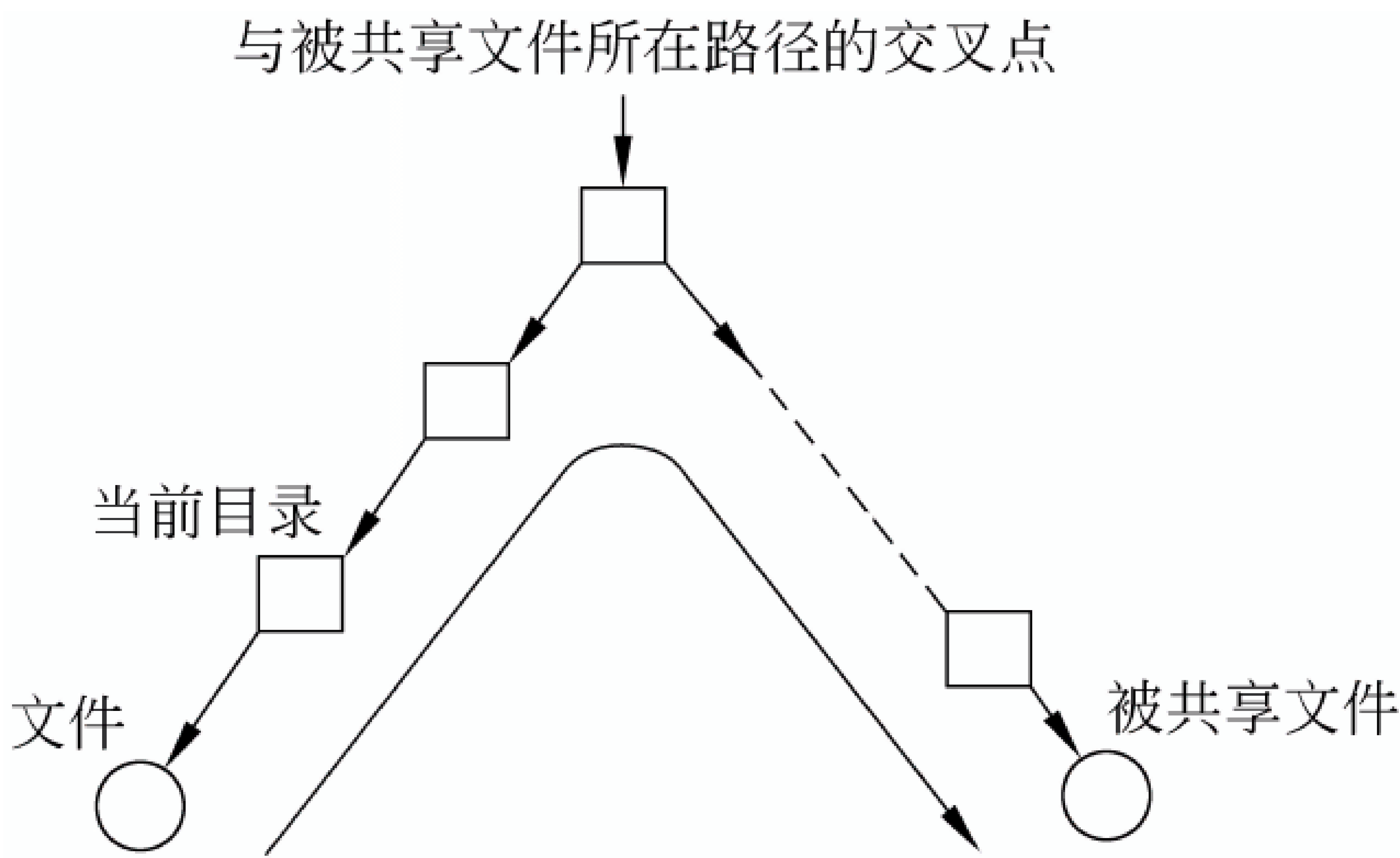



图 8.18　绕道法

显然,绕道法要绕弯路访问多级目录,从而其搜索效率不高。为了提高共享其他目录中文件的速度,另一种共享的办法是在相应目录表之间进行链接。即将一个目录中的链指针直接指向被共享文件所在的目录。链接法仍然需要用户指定被共享的文件和被链接的目录。

实现文件共享的一种有效方法是采用基本文件目录表(BFD)的方法。该方法把所有文件目录的内容分成两部分:一部分包括文件的结构信息、物理块号、存取控制和管理信息等,并由系统赋予唯一的内部标识符来标识;另一部分则由用户给出的符号名和系统赋给文件说明信息的内部标识符组成。这两部分分别称为符号文件目录表(SFD)和基本文件目录表(BFD)。SFD 中存放文件名和文件内部标识符,BFD 中存放除了文件名之外的文件说明信息和文件的内部标识符。这样组成的多级目录结构如图 8.19 所示。

在图 8.19 中,为了简单起见,未在 BFD 表项中列出结构信息、存取控制信息和管理控制信息等。另外,在文件系统中,系统通常预先规定赋予基本文件目录、空白文件目录和主目录(MFD)的符号文件目录固定不变的唯一标识符,在图中它们分别为 0、1、2。

采用基本文件目录方式可较方便地实现文件共享。如果用户要共享某个文件,则只需给出被共享的文件名,系统就会自动在 SFD 的有关文件处生成与被共享文件相同的内部标识符 ID。例如在图 8.19 中,用户 Wang 和 Zhang 共享标识符为 6 的文件,对于系统来说,标识符 6 指向同一个文件;而对 Wang 和 Zhang 两用户来说,则对应于不同的文件名 b. c 和 f. c。

### 8.5.4　目录管理

由上面讨论可知,存放文件说明信息或目录管理说明信息的目录项构成目录文件,这些文件同样存放在文件存储设备中。在存取一个文件时,必须访问多级目录。如果访问每级目录时都必须到文件存储设备上去搜索的话,这不仅大大浪费 CPU 的处理时间,降低了处理速度,而且还给输入输出设备增加了不应有的负担。一种解决办法是在系统初启时,把所

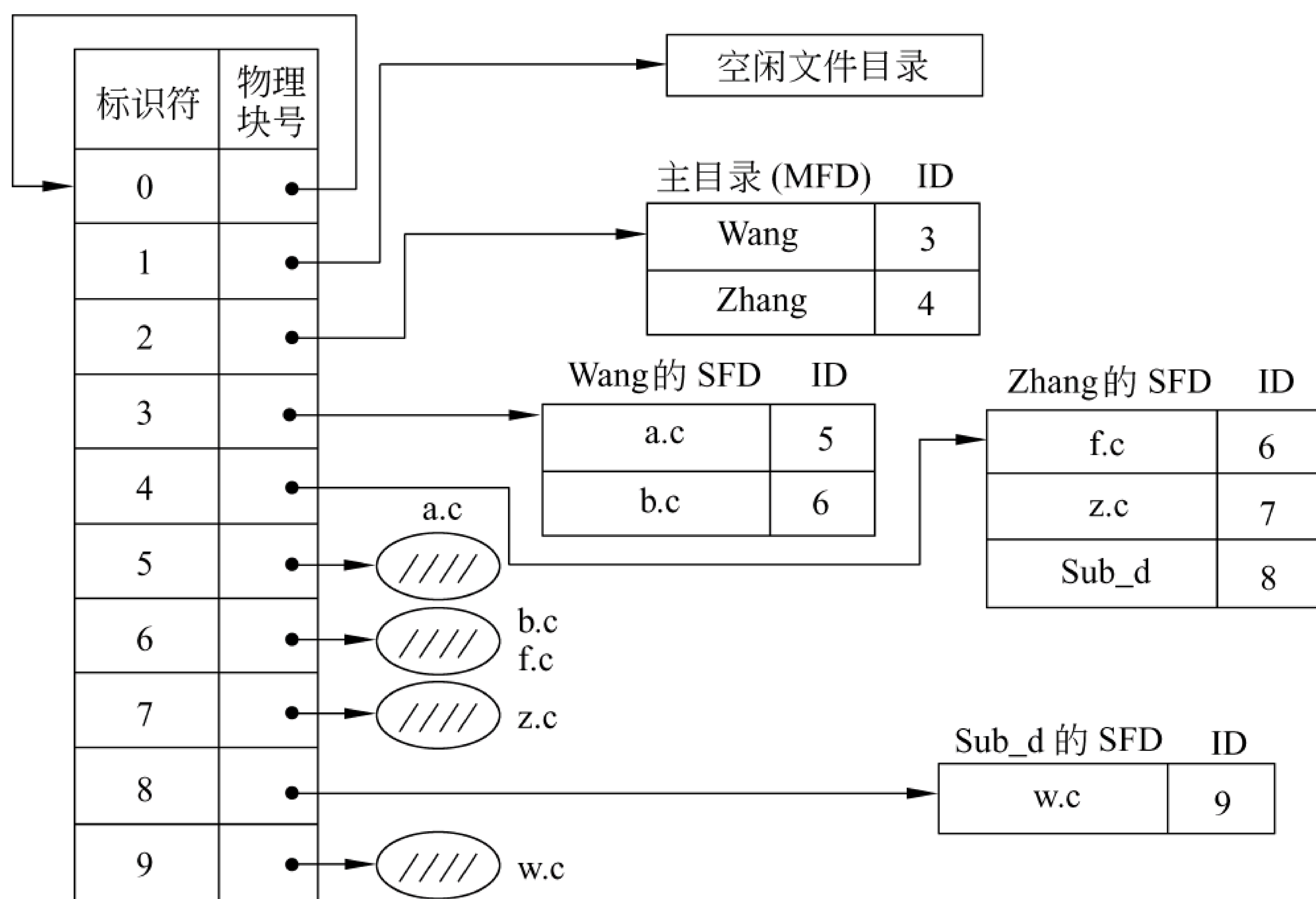



图 8.19　采用基本文件目录的多级目录结构

有的目录文件读入内存,由文件系统在内存完成对各级目录的搜索。这不仅减轻了输入输出设备的负担,而且由于内存访问速度高,因此,处理速度也将大大提高。不过,这种方法需要大量的内存支持,显然是不可取的。另一种折中的方法是:把当前正在使用的那些文件的目录表目复制到内存中,这样,既不占太多的内存容量,又可显著地减少搜索目录的时间和输入输出设备的压力。为此,系统提供两种特殊的操作,把有关的目录文件复制到内存的指定区,以及当用户不再访问有关信息文件时删除有关目录文件的内存副本。

把文件存储设备上的目录文件复制到内存的操作称为打开文件(fopen),而把删除文件的内存副本的操作称为关闭文件(fclose)。这两个操作一般以系统调用的方式提供给用户。对于按 BDF 和 SFD 方式排列的多级文件目录来说,系统按以下方式打开一个文件。

(1) 把主目录(MFD)中的相应表目,也就是与待打开文件相联系的有关表目复制到内存。例如,若准备打开图 8.19 中的文件 a.c,则将 MFD 中的第一项(Wang)复制到内存。

(2) 根据(1)所复制得到的标识符,再复制此标识符所指明的基本文件目录表(BDF)的有关表目,例如图 8.19 中的 ID=3 的 BDF 中的表目项。这个表目中包括有存取控制信息、结构信息以及下级目录的物理块号等。

(3) 根据(2)所得到的子目录说明信息搜索 SFD,以找到与待打开文件相对应的目录表项。如果找到的表目仍然是子目录名,则系统将根据其对应的标识符 ID 继续上述复制过程,直到所找到的表目是待打开的文件名,例如在图 8.19 中的文件名 a.c。

(4) 根据(3)所搜索到的文件名所对应的标识符 ID,把相应的 BDF 的表目项复制到内存。这样,待打开文件的说明信息就已复制到了内存中。由复制的文件说明,系统显然可以方便地得到文件的有关物理块号。从而,系统可对文件进行有关操作。

在完成了上述 4 个步骤之后,就说文件是被打开的了,称这样的文件为打开的文件或活动文件。而且,把内存中存放活动文件的 SFD 表目的表称为活动名字表,这个表每个用户一张。另外,把内存中存放活动文件的 BFD 表目的表称为活动文件表,这个表整个系统一张。

## 8.6 文件存取控制

本节介绍文件的存取控制问题。文件的存取控制是和文件的共享、保护和保密3个不同而又相互联系的问题紧密相关的。前面各节中简单地提及了文件的共享，但未涉及文件的保护和保密。

文件共享是指不同的用户共同使用一个文件。

文件保护则指文件本身需要防止文件的拥有者本人或其他用户破坏文件内容。

文件保密指未经文件拥有者许可，任何用户不得访问该文件。

这3个问题实际上是一个用户对文件的使用权限，即读、写、执行的许可权问题。

具体地说，文件系统的存取控制部分应做到：

(1) 对于拥有读、写或执行权限的用户，应让其对文件进行相应的操作。

(2) 对于没有读、写或执行权限的用户，应禁止他们对文件进行相应的操作。

(3) 应防止一个用户冒充其他用户对文件进行存取。

(4) 应防止拥有存取权限的用户误用文件。

这些功能是由一组称为存取控制验证模块的程序提供的。它们分3步验证用户的存取操作。

(1) 审定用户的存取权限。

(2) 比较用户权限与用户的本次存取要求是否一致。

(3) 将存取要求和被访问文件的保密性比较，看是否有冲突。

一般可有下述4个方式来验证用户的存取操作：

(1) 存取控制矩阵；

(2) 存取控制表；

(3) 口令；

(4) 密码术。

系统设计人员根据需要选择其中一种或几种并将相应的数据结构置于文件说明(BFD等)中，在用户访问文件时，对用户的存取权限、存取要求的一致性以及保密性等进行验证。下面简单地介绍这4种方式。

**1. 存取控制矩阵**

存取控制矩阵方式以一个二维矩阵来进行存取控制。二维矩阵的一维是所有的用户，另一维是所有的文件。对应的矩阵元素则是用户对文件的存取权限，包括读(R)、写(W)和执行(E)，如图8.20所示。

当用户向文件系统提出存取要求时，由存取控制验证模块根据该矩阵内容对本次存取要求进行比较，如果不匹配的话，系统拒绝执行。

存取控制矩阵的方法虽然在概念上比较简单，但是，当文件和用户较多时，存取控制矩阵将变得非常庞大，这无论是在占用内存空间的大小上，还是在为使用文件而对矩阵进行扫描的时间开销上都是不合适的。因此，在实现时往往采取某些辅助措施以减少时间和空间的开销。

| 文件名 \ 存取权限 \ 用户 | Wang | Lee | Zhang | … |
|---|---|---|---|---|
| A. C. | RWE | E | RWE | |
| B. C | RW | R | RWE | |
| D. C | R | W | WE | |
| E. C | R | W | RW | |

图 8.20　存取控制矩阵

| 用户 \ 文件名 | A. C |
|---|---|
| Zhang | RWE |
| A 组 | RE |
| B 组 | E |
| Wang | RWE |
| 其他 | None |

图 8.21　存取控制表

**2. 存取控制表**

存取控制表以文件为单位，把用户按某种关系划分为若干组，同时规定每组的存取权限。这样，所有用户组对文件权限的集合就形成了该文件的存取控制表，如图 8.21 所示。

每个文件都有一张存取控制表。在实现时，该表存放在文件说明中，也就是 BFD 的有关表目中。文件被打开时，由于存取控制表也相应地被复制到了内存活动文件中，因此，存取控制验证能高效地进行。

**3. 口令方式**

口令方式有两种。一种是当用户进入系统，为建立终端进程时获得系统使用权的口令。显然，如果用户输入的口令(password)与原来设置的口令不一致的话，该用户将被系统拒绝。另一种口令方式是，每个用户在创建文件时，为每一个创建的文件设置一个口令，且将其置于文件说明中。当任一用户想使用该文件时，都必须首先提供口令，只有当两者相符时才允许存取。显然，口令只有设置者自己知道，若允许其他用户使用自己的文件，口令设置者可将口令赋予其他用户。这样，既可以做到文件共享，又可做到保密。而且，由于口令较为简单，占用的内存单元以及验证口令所费时间都将非常少。不过，相对来说，口令方式保密性能较差。口令一旦被别人掌握，就可以获得与文件主同样的权利而没有任何等级差别，这就使得文件失窃的可能性大大增加。再者，当要修改某个用户的存取权限时，文件主必须修改口令，这样，所有共享该文件的用户的存取权限都被取消，除非文件主将新的口令通知用户。

**4. 密码方式**

防止文件泄密以及控制存取的另一种方法是密码方式。密码方式在用户创建源文件并将其写入存储设备时对文件进行编码加密，在读出文件时对其进行译码解密。显然，只有能够进行译码解密的用户才能读出被加密的文件信息，从而起到文件保密的作用。

文件的加密和解密都需要用户提供一个代码键(KEY)。加密程序根据这一代码键对用户文件进行编码变换，然后将其写入存储设备。在读取文件时，只有用户给定的代码键与加密时的代码键相一致时，解密程序才能对加密文件进行解密，将其还原为源文件。加密处理过程如图 8.22 所示。

加密方式具有保密性强的优点，因为与口令不同，进行编码解码的代码键没有存放在系统中，而是由用户自己掌握。但是，由于编码解码工作要耗费大量的处理时间，因此，加密技术是以牺牲系统开销为代价的。

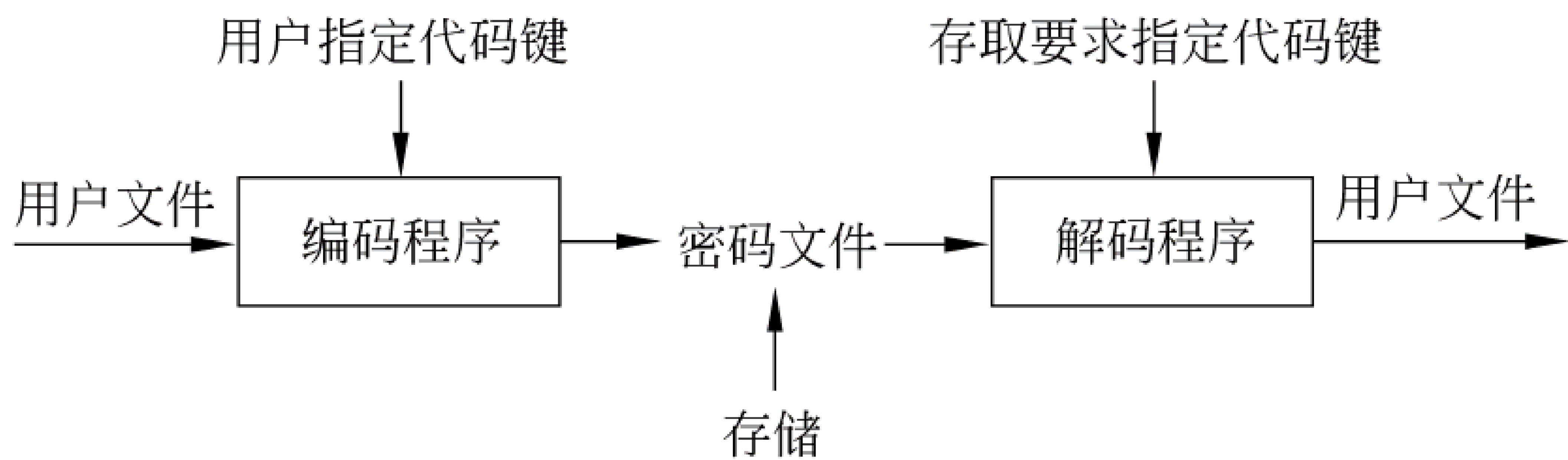



图 8.22　加密解密过程

## 8.7　文件的使用

上面各节主要从系统管理的角度讨论文件系统。本节讨论文件系统对用户的接口。

文件系统以系统调用方式或命令方式为用户提供下列几类服务：

(1) 关于设置和修改用户对文件的存取权限的服务；

(2) 关于建立、改变和删除目录的服务；

(3) 关于文件共享、设置访问路径的服务；

(4) 创建、打开、读写、关闭以及撤销文件的服务。

这些服务的调用名和参数都因系统而异。例如在 UNIX 系统中，chmod 命令可用来改变一个或多个文件或目录的读写控制模式。读者可在 UNIX 环境下使用命令 man chmod 命令阅读到 chmod 命令的全部详细信息。另外，mkdir、cd、rmdir 等命令则可用来建立、改变和删除指定的目录。有关这些命令的详细信息，同样可由 man 命令从 UNIX 系统中得到，这里不作介绍。

有关对文件操作的命令都基于操作系统提供的系统调用。这些系统调用包括建立文件用的 creat，读文件用的 read，关闭文件用的 close 以及撤销文件用的 delete 等。

其中，creat 调用将根据用户提供的文件名和属性，在指定的文件存储设备上建立一个文件并把文件标识符返回给用户。open 调用把在文件存储设备上的有关文件说明信息复制到内存的活动文件目录表中。write 调用将把从内存中某个位置开始的一段 $n$ 字节长(字符流文件时)信息或 $n$ 个记录经设备管理程序写入文件存储设备。read 调用的功能与 write 相反，它把指定文件的几个字节或记录读入内存中的指定区域。若文件暂时不用时，系统调用 close 关闭该文件。close 调用撤销活动文件表中相应的表目。delete 调用则在一个文件不再被访问时，删除该文件在文件存储设备上的有关说明信息，并释放该文件所占据的全部存储空间。

## 8.8　文件系统的层次模型

在上面各节中，从系统和用户两方面的角度讨论了文件系统的基本概念和功能。在本节中，将介绍文件系统的一般层次模型，以便使读者对文件系统形成一个完整的概念。

操作系统的层次结构的设计方法是 Dijkstra 于 1967 年提出的，1968 年 Madnick 将这一思想引入了文件系统。层次结构法的优点是，可以按照系统所提供的功能来划分为各种不同的层次，下层为上层提供服务，上层使用下层的功能。这样，上下层之间彼此无须了解

对方的内部结构和实现方法，而只关心二者的接口。从而，一个看上去十分复杂的系统将会由于层次的划分而变得易于设计、理解和实现。而且，当系统出现错误时，也容易进行查错和调整。因此，层次化设计方法也使得系统的管理和维护更加容易。

不过，在层次化设计方法中，层次的划分是一个十分复杂的问题。如果层次划分太少，则每一层的内容仍然十分复杂，分层的意义不明显；若层次划分太多，则各层之间传递的参数会急剧增加，而且每一层的处理会占去一定的系统开销，从而影响系统效率。因此，层次的划分要根据实际需要仔细地考虑。Madnick 把文件系统划分为 8 层，如图 8.23 所示。

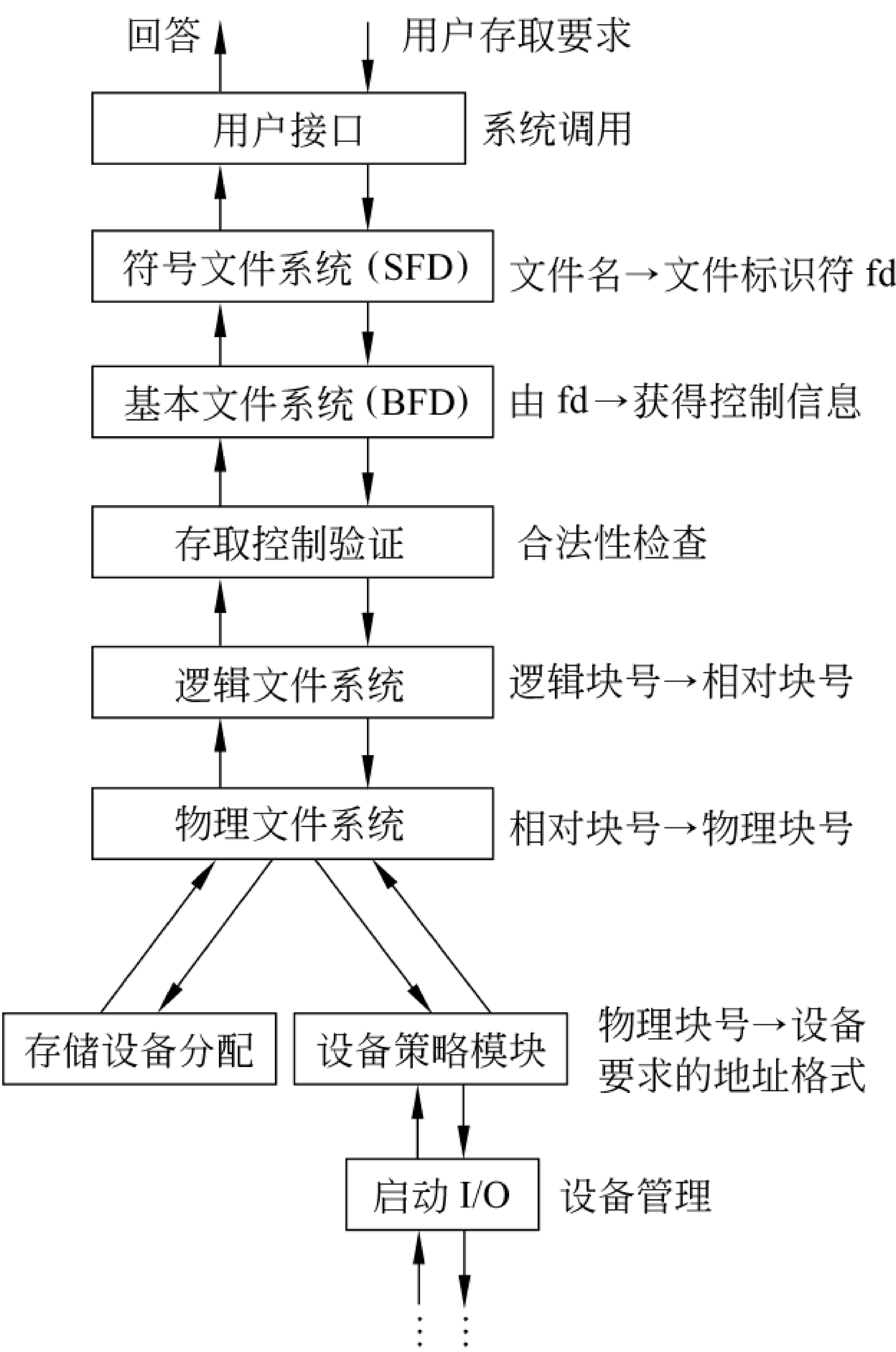



图 8.23　文件系统的层次模型

在图 8.23 中，第 1 层是用户接口，该层根据用户对文件的存取要求，把不同的系统调用加工改造成不同的内部调用格式。

第 2 层是符号文件系统层。该层完成第 1 层所提供的功能，并把第 1 层所提供的参数——用户文件名转换成系统内部的唯一标识符 fd，该层的主要工作是搜索文件目录，也就是搜索 SFD，以找到相应文件名的表目以找到 fd。然后，fd 将作为参数传给第 3 层。

第 3 层是基本文件系统层。该层根据第 2 层的调用参数 fd，找到文件的说明信息，包括存取控制表、文件逻辑结构、物理结构以及第一个物理块地址等。

第 4 层是存取控制验证层。该层的主要功能是根据存取控制信息和用户访问要求，检验文件访问的合法性，从而实现文件的共享、保护和保密。

第 5 层是逻辑文件系统层。该层的主要功能是根据文件的逻辑结构，找到所要进行操作的数据或记录的相对块号。对于字符流的无结构文件来说，只要把用户指定的逻辑地址

按块长换算成相对块号就可以了。但是，对于记录式有结构文件来说，由于用户有时指定的是关键字或记录名，因此，需首先由关键字(或记录名)搜索到相应的记录并得到对应的逻辑地址之后，再将其转换为相对块号。

第 6 层是物理文件系统层。该层把相对块号根据文件的物理结构转换成物理地址。

第 7 层是文件存储设备分配模块和设备策略模块。文件存储设备分配模块实现对空闲存储块的管理，包括分配、释放和组织。设备策略模块主要是把物理块号转换成相应文件存储设备所要求的地址格式，例如磁盘的柱面号、磁道号和盘区号等。然后，根据具体的操作要求及必要的参数，准备启动输入输出设备的命令。

第 8 层是启动输入输出层。由设备处理程序执行具体的读或写文件操作。

第 7 层和第 8 层是文件系统和设备管理程序的接口层。

## 本章小结

文件系统为用户提供了按名存取的功能，以使得用户能透明地存取文件。为了实现按名存取，文件需要对文件存储设备进行合理的组织、分配和管理；对存储在文件存储设备上的文件进行保护、保密和提供共享的手段。另外，文件系统还要提供检索文件或文件中记录的手段。文件系统就是完成上述功能的一组软件和数据结构的集合。

本章主要讨论了文件和文件系统的基本概念。文件是一组赋名的字符流的集合或一组相关联的记录的集合。一个记录是有意义的信息的基本单位，它有定长和变长两种基本格式。本章在定长的假定下讨论，但其结果也可以扩展到变长格式的情况。

为了合理有效地利用存储空间，以及高效率地进行按名存取，文件按一定的逻辑结构组成逻辑文件，逻辑文件是用户可见的抽象文件。文件的逻辑结构可分为字符流式无结构的连续文件、记录式有结构文件两大类。其中记录式文件又可分为连续结构、多重结构、转置结构及顺序结构文件等。对于记录式文件来说，如果用户在存取操作时指定的参数是关键字或记录名的话(按关键字存取)，有 3 种常用的方法可用来检索文件，它们是线性检索法、散列法和二分搜索法。

除了逻辑结构之外，一个文件在存储设备上按一定的物理方式存放。文件的物理结构受设备类型的影响。例如，磁带设备只适合连续存放和顺序存取，而磁盘设备既适合连续存放，也适合串联存放和索引存放。磁盘设备上的文件既可以是顺序存取的，也可以是直接存取或按关键字存取的。

当用户创建一个文件时，首先要给该文件分配足够的存储空间。存储空间的管理方法有空白文件目录、空闲块链和位示图法。比较有影响的存储空间管理方法是空闲块链中的成组链法。

文件名或记录名与物理地址之间的转换通过文件目录来实现。有单级目录、二级目录和多级目录几种目录结构。二级目录和多级目录是为了解决文件的重名问题和提高搜索速度而提出来的。多级目录构成文件树形结构。另外，为了便于共享，把目录项中存放的文件说明信息划分为两部分：文件内部标识符和文件名，存取控制信息以及结构信息等文件说明信息部分。前者的集合称为符号文件表(SFD)，后者的集合称为基本文件表(BFD)。

对文件的存取控制是和文件共享、保护和保密紧密相关的。存取控制可采用存取控制矩阵、存取控制表、口令和密码的方法进行存取验证，以确定用户权限。

最后，本章介绍了文件系统的使用方法和文件系统的层次模型。

## 习　　题

8.1　什么是文件和文件系统？文件系统有哪些功能？

8.2　文件一般根据什么分类？可以分为哪几类？

8.3　什么是文件的逻辑结构？什么是记录？

8.4　设文件的结构为多重结构和转置结构的组合，且定义函数 decode$(K,x)$和 retrive$(K,x)$如下：

函数 decode$(K,x)$为关键字 $K$ 的搜索函数。其中 $K$ 为待搜索关键字，$x$ 为顺序指针。当被搜索文件为多重结构时，指向关键字 $K$ 的指针只有一个，即 $e(K)$，此时 $x$＝nil，且 decode$(K,x)$返回指针 $e(K)$。当被搜索文件为转置结构时，一般指向关键字 $K$ 的指针有 $n$ 个，即 $e_1(K),\cdots,e_n(K)$，$n$ 为包含关键字 $K$ 的记录个数。此时，若 $x$＝nil，decode$(K,x)$返回 $e_1(K)$；若 $x=e_n(K)$，则 decode$(K,x)$返回 $nil$；否则，若 $x=e_i(K)$，$1\leqslant i<n$，则 decode$(K,x)$返回 $e_i+1(K)$。

函数 retrive$(K,x)$是记录搜索函数。其中 $K$ 为指向关键字的指针，$x$ 为记录顺序指针。如果 $x=nil$ 则 retrive$(K,x)$返回被搜索记录队列的第一个记录的逻辑地址；若 $x$ 等于该记录队列的最后一个记录的话，则 retrive$(K,x)$返回 nil；否则，retrive$(K,x)$返回 $x$ 的下一个记录的逻辑地址。

试问：

(1) 如果 decode$(K,x)$采用线性搜索法，怎样描述 decode$(K,x)$？

(2) 如果 retrive$(K,x)$采用二分搜索法，应怎样排列记录队列？怎样描述函数 retrive$(K,x)$？

(3) 设函数 search$(K)$给出含有关键字 $K$ 的所有记录的逻辑地址，试描述 search$(K)$。

8.5　设散列函数 $h(n)=(676\times l_1+26\times l_2+l_3)(\mathrm{mod}\ t)$，$t=11$，$l_i$ 为关键字 $n$ 的第 $i$ 个英文字母序号。关键字表长为 11，关键字为英文字母。先按线性散列法，再按平方散列法计算将任意 7 个关键字放入链表中所用的计算次数。这里令 $a=2$，$c=1$。

8.6　设关键字表按英文字母顺序排列，且两关键字项之间的距离为 $d$，写出 $n=3$ 的三分搜索算法。

8.7　文件的物理结构有哪几种？为什么说串联文件结构不适于随机存取？

8.8　设索引表长度为 13，其中 0～9 项为直接寻址方式，后 3 项为间接寻址方式，试描述出给定文件长度 $n$(块数)后的索引方式寻址算法。

8.9　常用的文件存储设备的管理方法有哪些？试述主要优缺点。

8.10　试述成组链法的基本原理，并描述成组链法的分配与释放过程。

8.11　什么是文件目录？文件目录中包含哪些信息？

8.12　二级目录和多级目录的好处是什么？符号文件目录表和基本文件目录表是二级目录吗？

8.13　文件存取控制方式有哪几种？试比较它们各自的优缺点。

8.14　设文件 SQRT 由连续结构的定长记录组成，每个记录的长度为 500B，每个物理块长 1000B，且物理结构也为连续结构和采用直接存取方式；试按照图 8.23 所示的文件系统模型，写出系统调用 Read（SQRT，5，15000）的各层执行结果。其中，SQRT 为文件名，5 为记录号，15000 为内存地址。

# 第 9 章　设备管理

设备管理是操作系统的重要组成部分之一。本章主要讨论设备管理的基本概念,包括中断、缓冲、设备分配和控制等。

## 9.1　引　　言

### 9.1.1　设备的类别

在计算机系统中,除了 CPU 和内存之外,其他的大部分硬设备称为外部设备。它包括常用的输入输出设备、外存设备以及终端设备等。这些设备种类繁多,特性各异,操作方式的区别也很大,从而使得操作系统的设备管理变得十分复杂。本节先从系统管理的角度将各种设备进行简单的分类,然后再介绍设备管理的主要功能与任务。

早期的计算机系统由于速度慢、应用面窄,外部设备主要以纸带、卡片等作为输入输出介质,相应的设备管理程序也比较简单。

进入 20 世纪 80 年代以来,由于个人计算机、工作站以及计算机网络系统等的发展,外部设备开始走向多样化、复杂化和智能化。例如,在有的网络卡中就装有自己的 CPU,以处理网络上信息的输入输出。再者,除了硬件设备之外,以某种硬件设备为基础的虚拟设备和仿真设备技术也得到了广泛应用。例如,虚终端技术和仿真终端技术等。实际上,近年来最为流行的窗口系统中的 X Window 等都是作为一种设备和操作系统相连的。这使得设备管理变得越来越复杂化。限于篇幅,本章只能介绍设备管理的基本原理和方法。掌握了这些基本原理和方法,读者在了解具体的设备管理系统时就会容易得多。

首先介绍设备的分类。按设备的使用特性可分为存储设备、输入输出设备、终端设备以及脱机设备等,如图 9.1 所示。

另外,按设备的从属关系,可把设备划分为系统设备和用户设备。系统设备是指那些在操作系统生成时就已配置好的各种标准设备,例如键盘、打印机以及文件存储设备等。而用户设备则是那些在系统生成时没有配置,而由用户自己安装配置后由操作系统统一管理的设备。例如,网络系统中的各种网板、实时系统中的 A/D 和 D/A 变换器以及图像处理系统的图像设备等。

对设备分类的目的在于简化设备管理程序。由于设备管理程序是和硬件打交道的,因此,不同的设备硬件对应于不同的管理程序。不过,对于同类设备来说,由于设备的硬件特性十分相似,从而可以利用相同的管理程序或只需做很少的修改即可。

除了上述分类方法之外,在有的系统中还按信息组织方式来划分设备。例如,UNIX 系统就把外部设备划分为字符设备和块设备。键盘、终端和打印机等以字符为单位组织和处理信息的设备被称为字符设备;而磁盘、磁带等以字符块为单位组织和处理信息的设备被称为块设备。

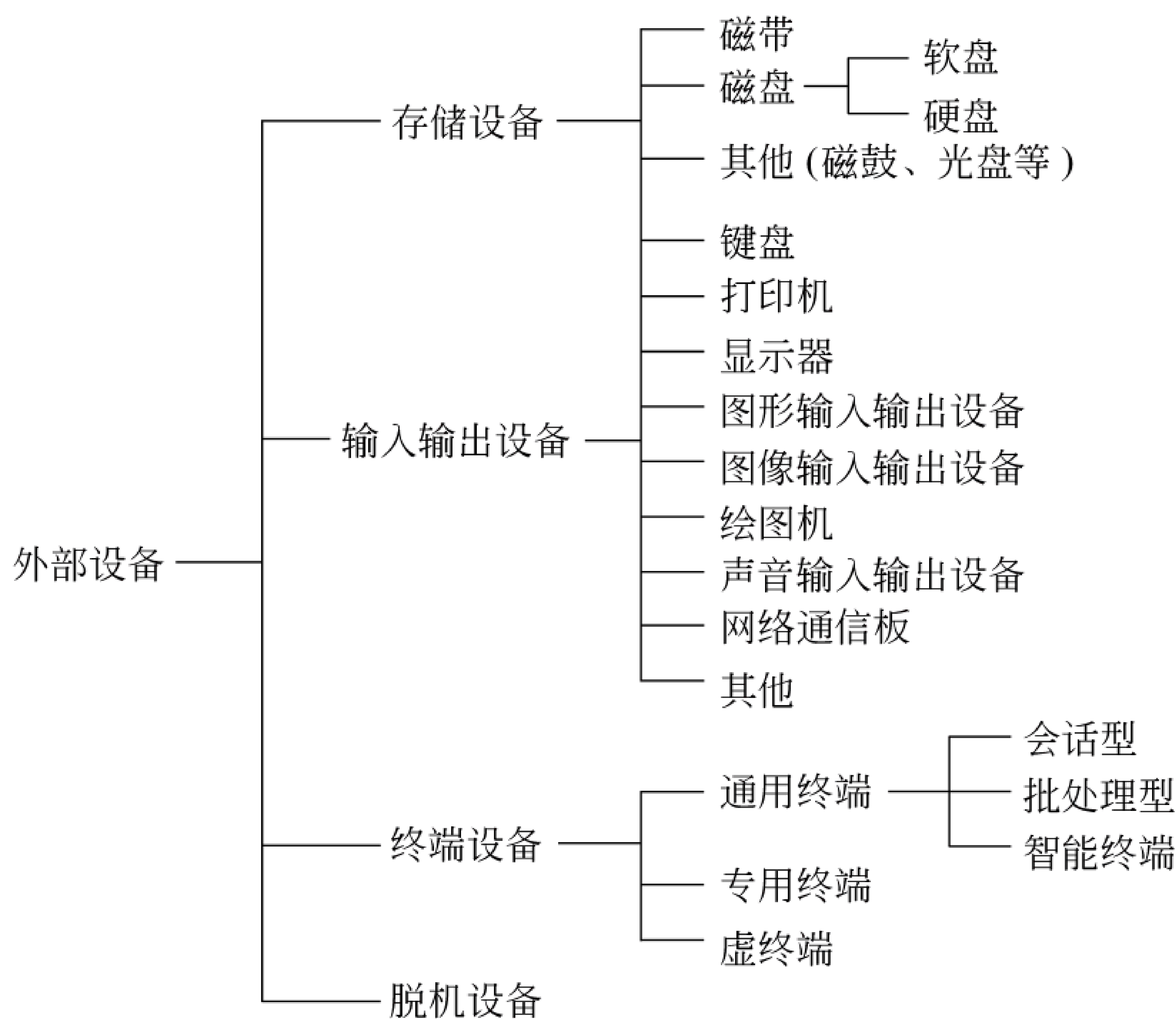



图 9.1　按使用特性对外部设备的分类

### 9.1.2　设备管理的功能和任务

设备管理是对计算机输入输出系统的管理,这是操作系统中最具有多样性和复杂性的部分。其主要任务如下:

(1) 选择和分配输入输出设备以便进行数据传输操作。

(2) 控制输入输出设备和 CPU(或内存)之间交换数据。

(3) 为用户提供一个友好的透明接口,把用户和设备硬件特性分开,使得用户在编制应用程序时不必涉及具体设备,系统按用户要求控制设备工作。另外,这个接口还为新增加的用户设备提供一个和系统核心相连接的入口,以便用户开发新的设备管理程序。

(4) 提高设备和设备之间、CPU 和设备之间,以及进程和进程之间的并行操作度,以使操作系统获得最佳效率。

为了完成上述主要任务,设备管理程序一般要提供下述功能:

(1) 提供和进程管理系统的接口。当进程要求设备资源时,该接口将进程要求转达给设备管理程序。

(2) 进行设备分配。按照设备类型和相应的分配算法把设备和其他有关的硬件分配给请求该设备的进程,并把未分配到所请求设备或其他有关硬件的进程放入等待队列。

(3) 实现设备和设备、设备和 CPU 等之间的并行操作。这需要有相应的硬件支持。除了装有控制状态寄存器、数据缓冲寄存器等的控制器之外,对应于不同的输入输出(I/O)控制方式,还需要有 DMA(Directed Memory Access)通道等硬件。从而,在设备分配程序根据进程要求分配了设备、控制器和通道(或 DMA)等硬件之后,通道(或 DMA)将自动完成设备和内存之间的数据传送工作,从而完成并行操作的任务。在没有通道(或 DMA)的系统里,则由设备管理程序利用中断技术来完成上述并行操作。

(4) 进行缓冲区管理。一般来说,CPU 的执行速度和访问内存速度都比较高,而外部

设备的数据流通速度则低得多(例如键盘),为了减少外部设备和内存与 CPU 之间的数据速度不匹配的问题,系统中一般设有缓冲区(器)来暂放数据。设备管理程序负责进行缓冲区分配、释放及有关的管理工作。

下面,首先介绍各种输入输出的控制方式,然后再介绍缓冲区管理、中断、陷阱以及软中断等基本概念。在此基础上,再介绍设备分配原则及有关分配算法,最后介绍 I/O 进程的概念及设备驱动过程。

## 9.2 数据传送控制方式

设备管理的主要任务之一是控制设备和内存或 CPU 之间的数据传送,本节介绍几种常用的数据传送控制方式。

选择和衡量控制方式有如下几条原则:

(1) 数据传送速度足够高,能满足用户的需要但又不丢失数据。

(2) 系统开销小,所需的处理控制程序少。

(3) 能充分发挥硬件资源的能力,使得 I/O 设备尽量忙,而 CPU 等待时间少。由计算机原理课,已经知道,为了控制 I/O 设备和内存之间的数据交换,每台外围设备都是按照一定的规律编码的。而且,设备和内存与 CPU 之间有相应的硬件接口支持同步控制、设备选择以及中断控制等。因此,假定本节的数据传送控制方式都是基于这些硬件基础的,从而不再讨论有关硬件部分。

外围设备和内存之间的常用数据传送控制方式有 4 种:

(1) 程序直接控制方式;

(2) 中断控制方式;

(3) DMA 方式;

(4) 通道方式。

下面分别给予介绍。

### 9.2.1 程序直接控制方式

程序直接控制方式(programmed direct control)就是由用户进程来直接控制内存或 CPU 和外围设备之间的信息传送。这种方式的控制者是用户进程。当用户进程需要数据时,它通过 CPU 发出启动设备准备数据的启动命令 Start,然后,用户进程进入测试等待状态。在等待时间内,CPU 不断地用一条测试指令检查描述外围设备的工作状态的控制状态寄存器。而外围设备只有将数据传送的准备工作做好之后,才将该寄存器置为完成状态。从而,当 CPU 检测到控制状态寄存器为完成状态,也就是该寄存器发出 Done 信号之后,设备开始往内存或 CPU 传送数据。反之,当用户进程需要向设备输出数据时,也必须同样发启动命令启动设备和等待设备准备好之后才能输出数据。除了控制状态寄存器之外,在 I/O 控制器中还有一类称为数据缓冲寄存器的寄存器。在 CPU 与外围设备之间传送数据时,输入设备每进行一次操作,首先把所输入的数据送入该寄存器,然后,CPU 再把其中的数据取走。反之,当 CPU 输出数据时,也是先把数据输出到该寄存器之后,再由输出设备将其取走。只有数据装入该寄存器之后,控制状态寄存器的值才会发生变化。程序直接控

制方式的控制流程如图 9.2 所示。

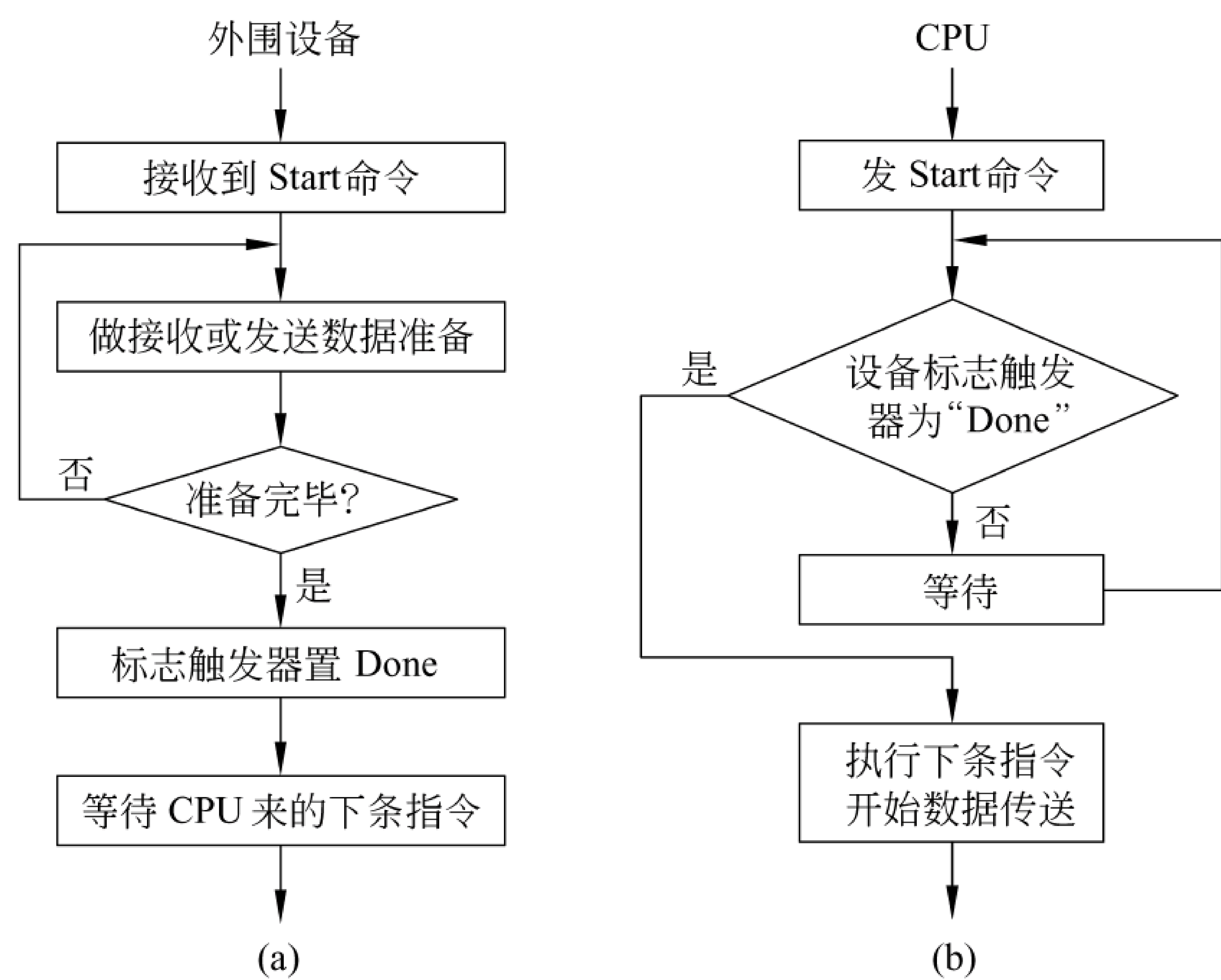



图 9.2　程序直接控制方式

程序直接控制方式虽然控制简单,也不需要多少硬件支持,但是,程序直接控制方式明显地存在下述缺点:

(1) CPU 和外围设备只能串行工作。由于 CPU 的处理速度要大大高于外围设备的数据传送和处理速度,所以,CPU 的大量时间都处于等待和空闲状态。这使得 CPU 的利用率大大降低。

(2) CPU 在一段时间内只能和一台外围设备交换数据信息,从而不能实现设备之间的并行工作。

(3) 由于程序直接控制方式依靠测试设备标志触发器的状态位来控制数据传送,因此无法发现和处理由于设备或其他硬件所产生的错误。所以,程序直接控制方式只适用于那些 CPU 执行速度较慢,而且外围设备较少的系统。

## 9.2.2　中断方式

为了减少程序直接控制方式中的 CPU 等待时间以及提高系统的并行工作程度,中断(interrupt)方式被用来控制外围设备和内存与 CPU 之间的数据传送。这种方式要求 CPU 与设备(或控制器)之间有相应的中断请求线,而且在设备控制器的控制状态寄存器有相应的中断允许位。中断方式的传送结构如图 9.3 所示。从而,数据的输入可按如下步骤操作。

(1) 进程需要数据时,通过 CPU 发出 Start 指令启动外围设备准备数据。该指令同时还将控制状态寄存器中的中断允许位打开,以便在需要时,中断程序可以被调用执行。

(2) 在进程发出指令启动设备之后,该进程放弃处理机,等待输入完成。从而,进程调度程序调度其他就绪进程占据处理机。

(3) 当输入完成时,I/O 控制器通过中断请求线向 CPU 发出中断信号。CPU 在接收到中断信号之后,转向预先设计好的中断处理程序对数据传送工作进行相应的处理。

(4) 在以后的某个时刻,进程调度程序选中提出请求并得到了数据的进程,该进程从约

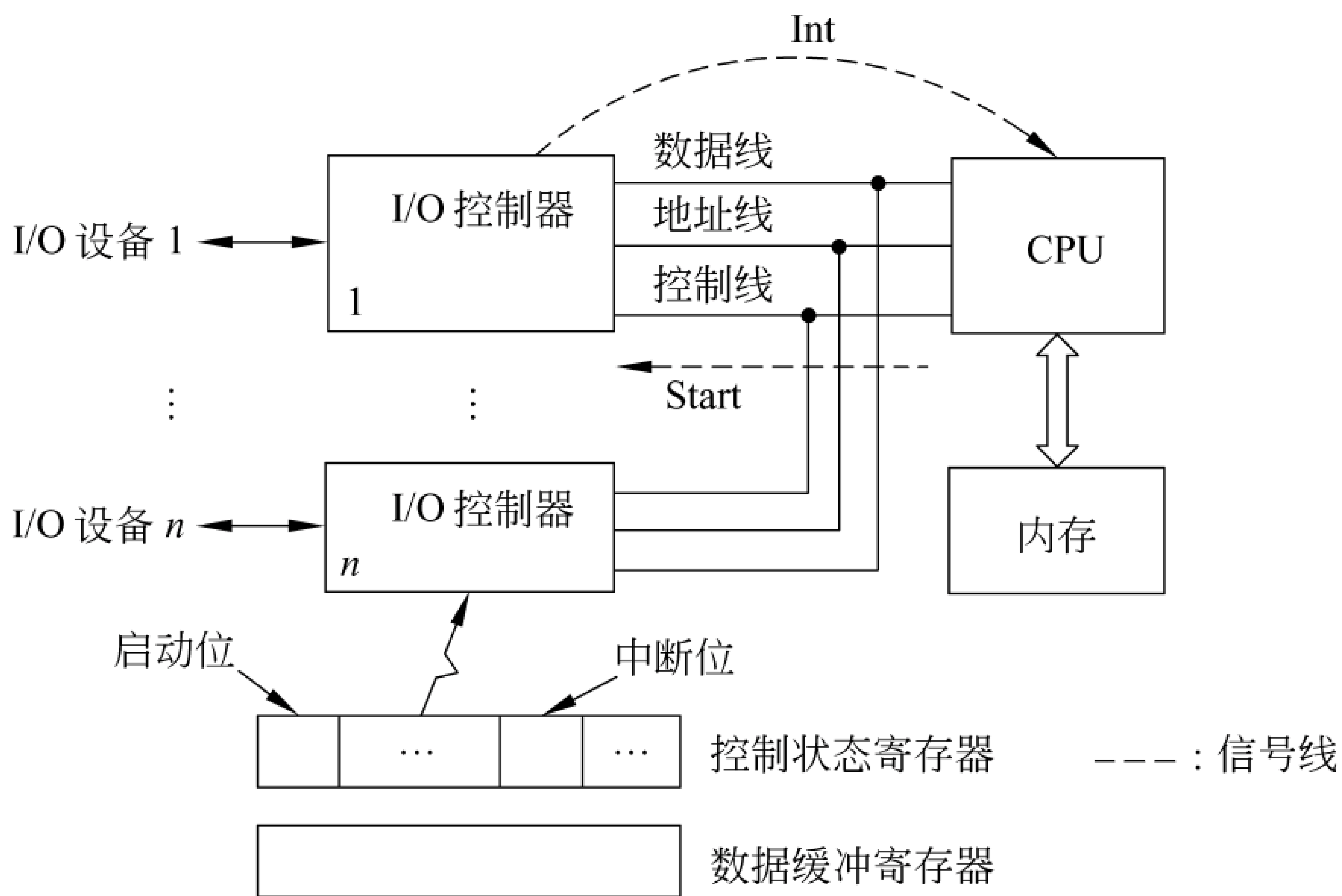



图 9.3　中断控制方式的传送结构

定的内存特定单元中取出数据继续工作。

中断控制方式的处理过程可由图 9.4 表示。

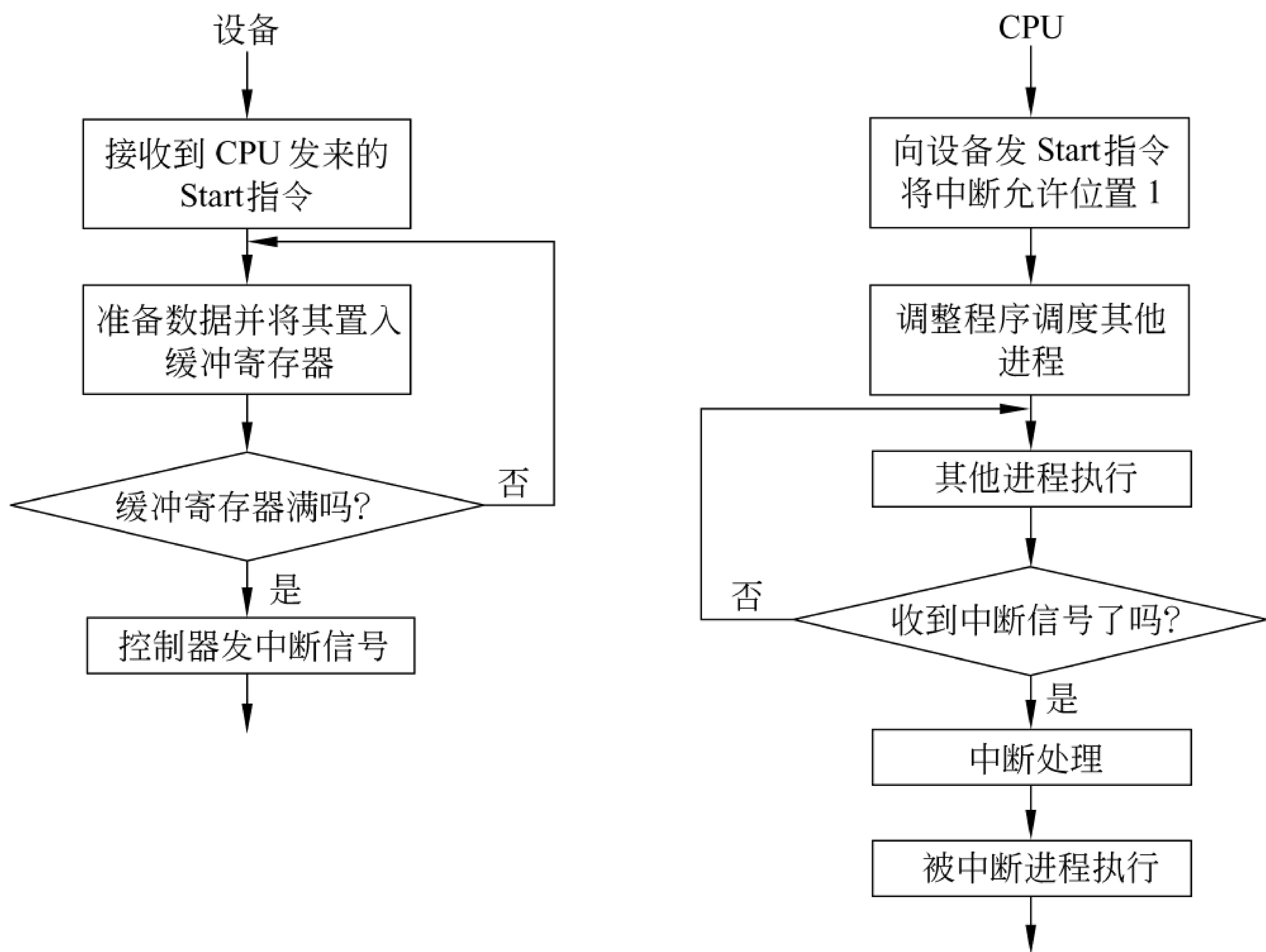



图 9.4　中断控制方式的处理过程

读者可以仿照上面的过程，描述输出情况下的中断控制方式的处理过程。由图 9.4 可以看出，当 CPU 发出启动设备和允许中断指令之后，它没有像程序直接控制方式那样循环测试状态控制寄存器的状态是否已处于 Done。反之，CPU 已被调度程序分配给其他进程在另外的进程上下文中执行。当设备将数据送入缓冲寄存器并发出中断信号之后，CPU 接收中断信号进行中断处理。显然，CPU 在另外的进程上下文中执行时，也可以发启动不同设备的启动指令和允许中断指令，从而做到设备与设备间的并行操作以及设备和 CPU 间的并行操作。

不过，尽管中断方式与程序直接控制方式相比，使 CPU 的利用率大大提高且能支持多道程序和设备的并行操作，但仍然存在着许多问题。首先，由于在 I/O 控制器的数据缓冲寄存器装满数据之后将会发生中断，而且数据缓冲寄存器通常较小，因此，在一次数据传送过程中，发生中断的次数较多。这将耗去大量的 CPU 处理时间。另外，现代计算机系统通常配置有各种各样的外围设备。如果这些设备通过中断处理方式进行并行操作，则由于中断次数的急剧增加而造成 CPU 无法响应中断和出现数据丢失现象。另外，在中断控制方式时，我们都是假定外围设备的速度非常低，而 CPU 处理速度非常高。也就是说，当设备把数据放入数据缓冲寄存器并发出中断信号之后，CPU 有足够的时间在下一个（组）数据进入数据缓冲寄存器之前取走这些数据。如果外围设备的速度也非常高，则可能造成数据缓冲寄存器的数据由于 CPU 来不及取走而丢失。DMA 方式和通道方式不会造成上述问题。

### 9.2.3 DMA 方式

DMA（Direct Memory Access）方式又称直接存取方式。其基本思想是在外围设备和内存之间开辟直接的数据交换通路。在 DMA 方式中，I/O 控制器具有比中断方式和程序直接控制方式时更强的功能。另外，除了控制状态寄存器和数据缓冲寄存器之外，DMA 控制器中还包括传送字节计数器、内存地址寄存器等。这是因为 DMA 方式窃取或挪用 CPU 的一个工作周期把数据缓冲寄存器中的数据直接送到内存地址寄存器所指向的内存区域中的缘故。

从而，DMA 控制器可用来代替 CPU 控制内存和设备之间进行成批的数据交换。批量数据（数据块）的传送由计数器逐个计数，并由内存地址寄存器确定内存地址。除了在数据块传送开始时需要 CPU 的启动指令和在整个数据块传送结束时需发中断通知 CPU 进行中断处理之外，不再像中断控制方式时那样需要 CPU 的频繁干预。DMA 方式的结构如图 9.5 所示。

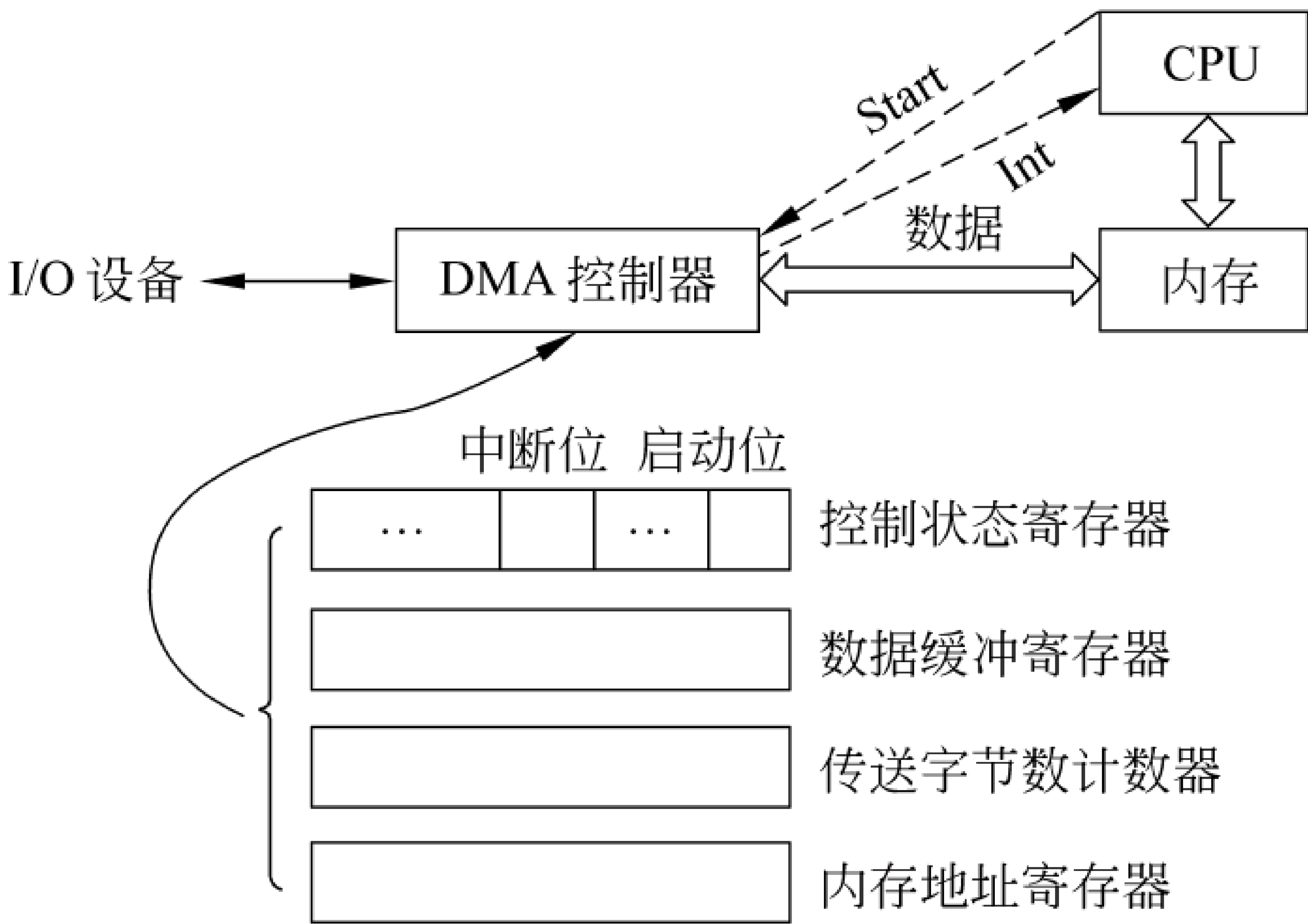



图 9.5　DMA 方式的传送结构

DMA 方式的数据输入处理过程如下：

(1) 当进程要求设备输入数据时，CPU 把准备存放输入数据的内存始址以及要传送的字节数分别送入 DMA 控制器中的内存地址寄存器和传送字节数计数器，另外，还把控制状态寄存器中的中断允许位和启动位置 1，从而启动设备开始进行数据输入。

(2) 发出数据要求的进程进入等待状态，进程调度程序调度其他进程占据 CPU。

(3) 输入设备不断地挪用 CPU 工作周期，将数据缓冲寄存器中的数据源源不断地写入内存，直到所要求的字节全部传送完毕。

(4) DMA 控制器在传送字节数完成时通过中断请求线发出中断信号，CPU 在接收到中断信号后转中断处理程序进行善后处理。

(5) 中断处理结束时，CPU 返回被中断进程处执行或被调度到新的进程上下文环境中执行。

DMA 方式的数据传送处理过程如图 9.6 所示。

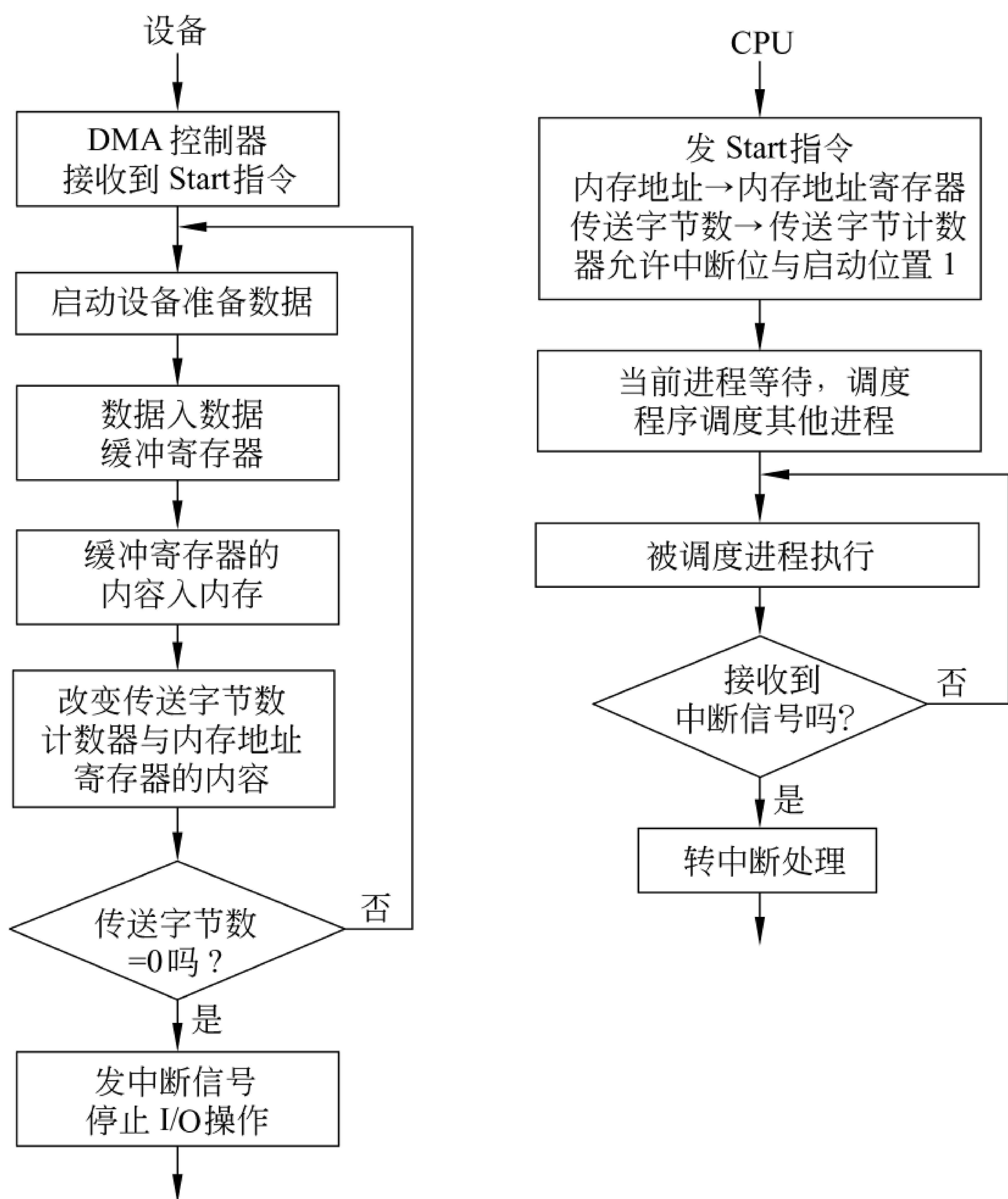



图 9.6　DMA 方式的数据传送处理过程

由图 9.6 可以看出，DMA 方式与中断方式的一个主要区别是，中断方式时是在数据缓冲寄存器满之后发中断要求 CPU 进行中断处理，而 DMA 方式则是在所要求传送的数据块全部传送结束时要求 CPU 进行中断处理。这就大大减少了 CPU 进行中断处理的次数。另一个主要区别是，中断方式的数据传送是在中断处理时由 CPU 控制完成的，而 DMA 方式是在 DMA 控制器的控制下不经过 CPU 控制完成的。这就排除了因并行操作设备过多时 CPU 来不及处理或因速度不匹配而造成数据丢失等现象。

不过，DMA 方式仍存在着一定的局限性。首先，DMA 方式对外围设备的管理和某些操作仍由 CPU 控制。在大中型计算机中，系统所配置的外设种类越来越多，数量也越来越大，因而，对外围设备的管理的控制也就愈来愈复杂。多个 DMA 控制器的同时使用显然会引起内存地址的冲突并使得控制过程进一步复杂化。同时，多个 DMA 控制器的同时使用

也是不经济的。因此，在大中型计算机系统中（近年来甚至在那些要求 I/O 能力强的微机系统中，例如在 Compaq 的 System pro386 系列微机系统中），除了设置 DMA 器件之外，还设置专门的硬件装置——通道。下面介绍通道控制方式。

### 9.2.4 通道控制方式

通道控制（channel control）方式与 DMA 方式相似，也是一种以内存为中心，实现设备和内存直接交换数据的控制方式。在 DMA 方式中，数据的传送方向、存放数据的内存始址以及传送的数据块长度等都由 CPU 控制，而在通道方式中，这些都由专管输入输出的硬件——通道来进行控制。另外，与 DMA 方式时每台设备至少一个 DMA 控制器相比，通道控制方式可以做到一个通道控制多台设备与内存进行数据交换，从而，通道方式进一步减轻了 CPU 的工作负担和增加了计算机系统的并行工作程度。

由于通道是一个专管输入输出操作控制的硬件，有必要更进一步完整地描述一下通道的定义：通道是一个独立于 CPU 的专管输入输出控制的处理机，它控制设备与内存直接进行数据交换。它有自己的通道指令，这些通道指令由 CPU 启动，并在操作结束时向 CPU 发中断信号。

通道的定义给出了通道控制方式的基本思想。在通道控制方式中，I/O 控制器中没有传送字节计数器和内存地址寄存器；但多了通道设备控制器和指令执行机构。在通道方式下，CPU 只需发出启动指令，指出通道相应的操作和 I/O 设备，该指令就可启动通道并使该通道从内存中调出相应的通道指令执行。

通道指令一般包含被交换数据在内存中应占据的位置、传送方向、数据块长度以及被控制的 I/O 设备的地址信息、特征信息（例如是磁带设备还是磁盘设备）等，通道指令在通道中没有存储部件时存放在内存中。

通道指令的格式一般由操作码（读、写或控制）、计数段（数据块长度）以及内存地址段和结束标志等组成。通道指令在进程要求数据时由系统自动生成。例如：

```
write 0 0 250 1850
write 1 1 250 720
```

是两条把一个记录的 500 个字符分别写入从内存地址 1850 开始的 250 个单元和从内存地址 720 开始的 250 个单元中。其中假定 write 操作码后的 1 是通道指令结束标志，而另一个 1 则是记录结束标志。该指令中省略了设备号和设备特征。

另外，一个通道可以以分时方式同时执行几个通道指令程序。按照信息交换方式的不同，一个系统中可设立 3 种类型的通道，即字节多路通道、数组多路通道和选择通道。由这 3 种通道组成的数据传送控制结构如图 9.7 所示。

字节多路通道以字节为单位传送数据，它主要用来连接大量的低速设备，如终端、打印机等。

数组多路通道以块为单位传送数据，它具有传送速率高和能分时操作不同的设备等优点。数组多路通道主要用来连接中速块设备，如磁带机等。

数组多路通道和字节多路通道都可以分时执行不同的通道指令程序。但是选择通道一次只能执行一个通道指令程序，所以选择通道一次只能控制一台设备进行 I/O 操作。不

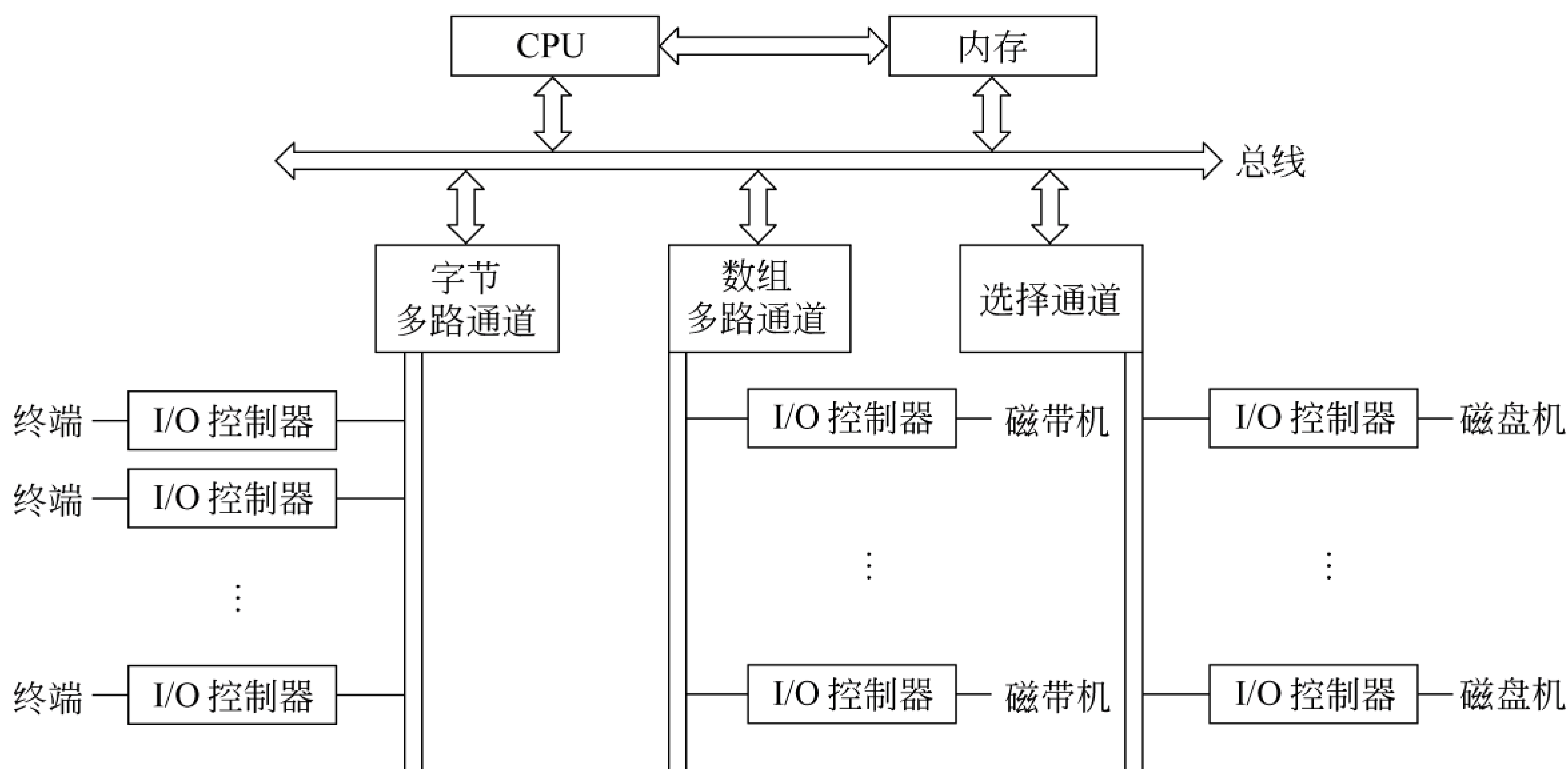



图 9.7　通道方式的数据传送结构

过，选择通道具有传送速度高的特点，因而它被用来连接高速外部设备，并以块为单位成批传送数据。受选择通道控制的外设有磁盘机等。

通道控制方式的数据输入处理过程可描述如下：

(1) 当进程要求设备输入数据时，CPU 发 Start 指令指明 I/O 操作、设备号和对应通道。

(2) 对应通道接收到 CPU 发来的启动指令 Start 之后，把存放在内存中的通道指令程序读出，设置对应设备的 I/O 控制器中的控制状态寄存器。

(3) 设备根据通道指令的要求，把数据送往内存中的指定区域。

(4) 若数据传送结束，I/O 控制器通过中断请求线发中断信号，请求 CPU 做中断处理。

(5) 与 DMA 方式时相同，即中断处理结束后 CPU 返回被中断进程处继续执行。

在(1)中要求数据的进程只有在调度程序选中它之后，才能对所得到的数据进行加工处理。

读者可以仿照 DMA 方式的 CPU 和设备的处理过程图(见图 9.6)画出通道控制方式的 CPU 和设备的处理流程图。

另外，在许多情况下，人们可从 CPU 执行的角度描述中断控制方式、DMA 方式或通道控制方式的控制处理过程。作为一个例子，这里给出通道控制方式的描述过程。

```
Channel control procedure:
    repeat
        IR←M[pc]
        pc←pc+1
        execute(IR)
        if require accessing with I/O Device
        then Command(I/O operation,Address of I/O device,channel) fi
        if I/O Done interrupt
        then Call interrupt processing control fi
    until machine halt
Interrupt processing control procedure
```

…

其中，IR 代表指令寄存器，pc 代表程序计数器，而 fi 则表示 if…then…条件语句的结束。关于 interrupt processing control 部分，在 9.3 节中将进一步讨论，这里暂时不做介绍。

## 9.3 中断技术

从 9.2 节可以看出，除了程序直接控制方式之外，无论是中断控制方式、DMA 方式还是通道控制方式，都需在设备和 CPU 之间进行通信，由设备向 CPU 发中断信号之后，CPU 接收相应的中断信号进行处理。这几种方式的区别只是中断处理的次数、数据传送方式以及控制指令的执行方式等。在计算机系统中，除了上述 I/O 中断之外，还存在着许多其他的突发事件，例如电源掉电、程序出错等，这些也会发出中断信号通知 CPU 做相应的处理。本节进一步讨论中断问题。

### 9.3.1 中断的基本概念

中断(interrupt)是指计算机在执行程序期间，系统内发生任何非寻常的或非预期的急需处理事件，使得 CPU 暂时中断当前正在执行的程序而转去执行相应的事件处理程序，待处理完毕后又返回原来被中断处继续执行或调度新的进程执行的过程。引起中断发生的事件被称为中断源。中断源向 CPU 发出的请求中断处理信号称为中断请求，而 CPU 收到中断请求后转相应的事件处理程序称为中断响应。

在有些情况下，尽管产生了中断源和发出了中断请求，但 CPU 内部的处理机状态字(PSW)的中断允许位已被清除，从而不允许 CPU 响应中断。这种情况称为禁止中断。CPU 禁止中断后只有等到 PSW 的中断允许位被重新设置后才能接收中断。禁止中断也称为关中断，PSW 的中断允许位的设置也被称为开中断。由计算机原理课可知中断请求、关中断和开中断等都是由硬件实现的。

开中断和关中断是为了保证某些程序执行的原子性。

除了禁止中断的概念之外，还有一个比较常用的概念是中断屏蔽。中断屏蔽是指在中断请求产生之后，系统用软件方式有选择地封锁部分中断而允许其余部分的中断仍能得到响应。

中断屏蔽是通过每一类中断源设置一个中断屏蔽触发器来屏蔽它们的中断请求而实现的。不过，有些中断请求是不能屏蔽甚至不能禁止的，也就是说，这些中断具有最高优先级。不管 CPU 是否是关中断的，只要这些中断请求一旦提出，CPU 必须立即响应。例如，电源掉电事件所引起的中断就是不可禁止和屏蔽中断。

### 9.3.2 中断的分类与优先级

根据系统对中断处理的需要，操作系统一般对中断进行分类，并对不同的中断赋予不同的处理优先级，以便在不同的中断同时发生时，按轻重缓急进行处理。

根据中断源产生的条件，可把中断分为外中断和内中断。

外中断指来自处理机和内存外部的中断，包括 I/O 设备发出的 I/O 中断、外部信号中断(例如用户按 Esc 键)、各种定时器引起的时钟中断以及调试程序中设置的断点等引起的

调试中断等。外中断在狭义上一般被称为中断。

内中断主要指在处理机和内存内部产生的中断。内中断一般称为陷阱(trap)。它包括程序运算引起的各种错误,如地址非法、校验错、页面失效、存取访问控制错、算术操作溢出、数据格式非法、除数为零、非法指令、用户程序执行特权指令、分时系统中的时间片中断以及从用户态到核心态的切换等都是陷阱的例子。

为了按中断源的轻重缓急处理响应中断,操作系统对不同的中断赋予不同的优先级。例如,在 UNIX 系统中,外中断和陷阱的优先级共分为 8 级。为了禁止中断或屏蔽中断,CPU 的处理机状态字(PSW)中也设置有相应的优先级。如果中断源的优先级高于 PSW 的优先级,则 CPU 响应该中断源的中断请求,反之,CPU 屏蔽该中断源的中断请求。

各中断源的优先级在系统设计时给定,在系统运行时是固定的。而处理机的优先级则根据执行情况由系统程序动态设定。

除了在优先级的设置方面有区别之外,中断和陷阱还有如下主要区别:

(1) 陷阱通常由处理机正在执行的现行指令引起,而中断则是由与现行指令无关的中断源引起的。

(2) 陷阱处理程序提供的服务为当前进程所用,而中断处理程序提供的服务则不是为了当前进程的。

(3) CPU 在执行完一条指令之后、下一条指令开始之前响应中断,而在一条指令执行中也可以响应陷阱。例如执行指令非法时,尽管被执行的非法指令不能执行结束,但 CPU 仍可对其进行处理。

另外,在有的系统中,陷阱处理程序被规定在各自的进程上下文中执行,而中断处理程序则在系统上下文中执行。

### 9.3.3 软中断

上述中断和陷阱都可以看作是硬中断,因为这些中断和陷阱要通过硬件产生相应的中断请求。而软中断则不然,它是通信进程之间用来模拟硬中断的一种信号通信方式。软中断与硬中断相同的地方是:其中断源发中断请求或软中断信号后,CPU 或接收进程在适当的时机自动进行中断处理或完成软中断信号所对应的功能。这里用“适当的时机”几个字是表示接收软中断信号的进程不一定正好在接收时占有处理机,而相应的处理必须等到该接收进程得到处理机之后才能进行。如果该接收进程是占据处理机的,那么,与中断处理相同,该接收进程在接收到软中断信号后将立即转去执行该软中断信号所对应的功能。

软中断的概念主要来源于 UNIX 系统。在前面介绍进程通信的有关章节中,已对 UNIX 的软中断通信进行了介绍,这里不再重复。

需要说明的一点是,在有些系统中,大部分的陷阱是转化为软中断处理的。由于陷阱主要与当前执行进程有关,因此,如果当前执行指令产生陷阱的话,则向当前执行进程自身发出一个软中断信号从而立即进入陷阱处理程序。

### 9.3.4 中断处理过程

一旦 CPU 响应中断,转入中断处理程序,系统就开始进行中断处理。下面说明中断处理过程:

(1) CPU 检查响应中断的条件是否满足。CPU 响应中断的条件是有来自中断源的中断请求且 CPU 允许中断。如果中断响应条件不满足,则中断处理无法进行。

(2) 如果 CPU 响应中断,则 CPU 关中断,使其进入不可再次响应中断的状态。

(3) 保存被中断进程现场。为了在中断处理结束后能使进程正确地返回到中断点,系统必须保存当前处理机状态字(PSW)和程序计数器(PC)等的值。这些值一般保存在特定堆栈或硬件寄存器中。

(4) 分析中断原因,调用中断处理子程序。在多个中断请求同时发生时,处理优先级最高的中断源发出的中断请求。

在系统中,为了方便处理,通常都是针对不同的中断源编制有不同的中断处理子程序(陷阱处理子程序)。这些子程序的入口地址(或陷阱指令的入口地址)存放在内存的特定单元中。再者,不同的中断源也对应着不同的处理机状态字(PSW)。这些不同的 PSW 被放在相应的内存单元中。存放的 PSW 与中断处理子程序入口地址一起构成中断向量。显然,根据中断或陷阱的种类,系统可由中断向量表迅速地找到该中断响应的优先级、中断处理子程序(或陷阱指令)的入口地址和对应的 PSW。

(5) 执行中断处理子程序。对陷阱来说,在有些系统中则是通过陷阱指令向当前执行进程发软中断信号后调用对应的处理子程序执行。

(6) 退出中断,恢复被中断进程的现场或调度新进程占据处理机。

(7) 开中断,CPU 继续执行。

中断处理过程如图 9.8 所示。

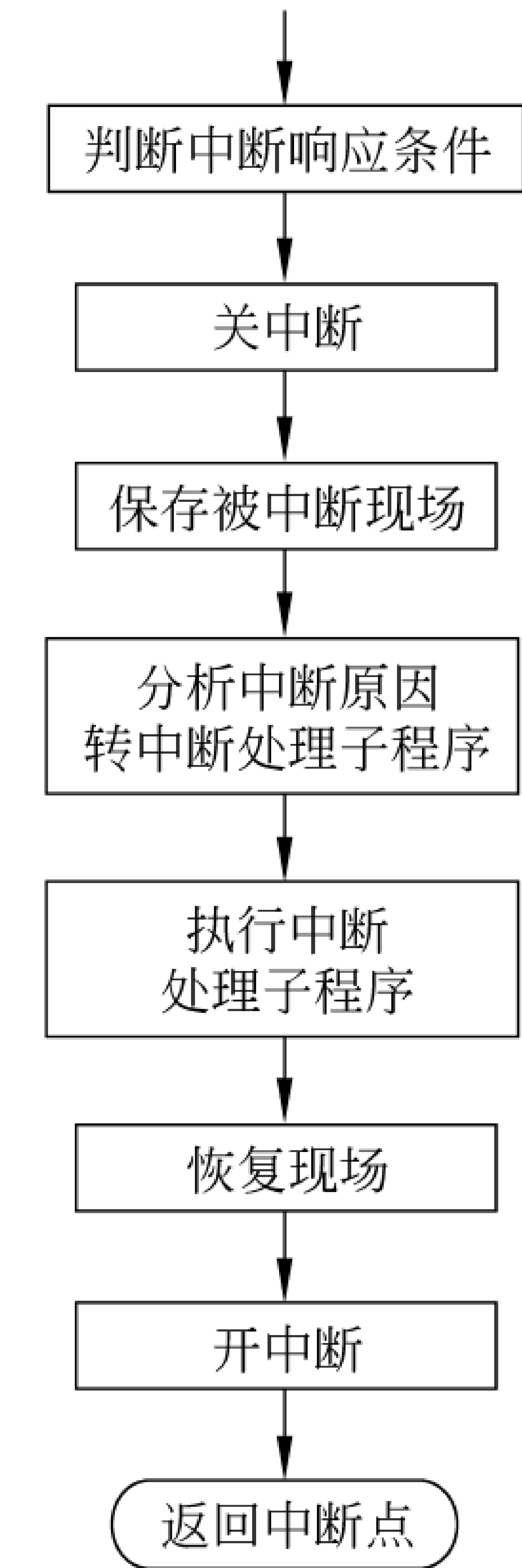


图 9.8 中断处理过程

有些系统中只在保存和恢复现场时禁止中断,而在执行中断处理子程序时屏蔽中断。

上面描述了中断处理过程的各个步骤。下面从 CPU 处理的角度出发来形式化地描述 I/O 中断处理的控制过程,以期读者能对中断处理过程有更深入的了解。

```
I/O interrupt processing control:
    begin
        unusable I/O interrupt flag
        save status of interrupt program
        if input Device i Ready
        then Call input Device i Control fi
        if Output Device i Ready
        then Call Output Device i Control fi
        if Data Deliver Done
        then Call Data Deliver Done Control fi
        restore CPU status
        reset I/O interrupt flag
    end
```

```
Input Device i Control: …
Output Device i Control: …
Data Deliver Done Control: …
```

## 9.4 缓冲技术

### 9.4.1 缓冲的引入

虽然中断、DMA 和通道控制技术使得系统中设备和设备、设备和 CPU 等得以并行工作，但是，正如在前面几节所讲述的那样，外围设备和 CPU 的处理速度不匹配的问题是客观存在的。这限制了和处理机连接的外设台数，且在中断方式时造成数据丢失。从而，外围设备和 CPU 处理速度不匹配的问题极大地制约了计算机系统性能的进一步提高和限制了系统的应用范围。

例如，当计算进程阵发性地把大批量数据输出到打印机上打印时，由于 CPU 输出数据的速度大大高于打印机的打印速度，因此，CPU 只好停下来等待。反之，在计算进程进行计算时，打印机又因无数据输出而空闲无事。

外围设备与处理机速度不匹配的问题可以采用设置缓冲区(器)的方法解决。在设置了缓冲区之后，计算进程可把数据首先输出到缓冲区，然后继续执行；而打印机则可以从缓冲区取出数据慢慢打印。

再者，从减少中断的次数看，也存在着引入缓冲区的必要性。在中断方式时，如果在 I/O 控制器中增加一个 100 个字符的缓冲器，则由前面对中断方式的描述可知，I/O 控制器对处理机的中断次数将降低 100 倍，即等到能存放 100 个字符的字符缓冲区装满之后才向处理机发一次中断。这将大大减少处理机的中断处理时间。即使是使用 DMA 方式或通道方式控制数据传送时，如果不划分专用的内存区或专用缓冲器来存放数据的话，也会因为要求数据的进程所拥有的内存区不够或存放数据的内存始址计算困难等原因，而造成某个进程长期占有通道或 DMA 控制器及设备，从而产生所谓瓶颈问题。

因此，为了匹配外设与 CPU 之间的处理速度，为了减少中断次数和 CPU 的中断处理时间，同时也是为了解决 DMA 或通道方式时的瓶颈问题，在设备管理中引入了用来暂存数据的缓冲技术。

根据 I/O 控制方式，缓冲的实现方法有两种，一种是采用专用硬件缓冲器，例如 I/O 控制器中的数据缓冲寄存器；另一种方法是在内存中划出一个具有 $n$ 个单元的专用缓冲区，以便存放输入输出的数据。内存缓冲区又称软件缓冲。

### 9.4.2 缓冲的种类

根据系统设置的缓冲器的个数，可把缓冲技术分为单缓冲、双缓冲和多缓冲以及缓冲池几种。

单缓冲是在设备和处理机之间设置一个缓冲器。设备和处理机交换数据时，先把被交换数据写入缓冲器，然后，需要数据的设备或处理机从缓冲器取走数据。由于缓冲器属于临界资源，即不允许多个进程同时对一个缓冲器操作，因此，尽管单缓冲能匹配设备和处理机

的处理速度，但是，设备和设备之间不能通过单缓冲达到并行操作。

解决两台外设、打印机和终端之间的并行操作问题的办法是设置双缓冲。有了两个缓冲器之后，CPU 可把输出到打印机的数据放入其中一个缓冲器（区），让打印机慢慢打印；然后，它又可以从另一个为终端设置的缓冲器（区）中读取所需要的输入数据。

显然，双缓冲只是一种说明设备和设备、CPU 和设备并行操作的简单模型，并不能用于实际系统中的并行操作。这是因为计算机系统中的外围设备较多，另外，双缓冲也很难匹配设备和处理机的处理速度。因此，现代计算机系统中一般使用多缓冲或缓冲池结构。

多缓冲是把多个缓冲区连接起来组成两部分，一部分专门用于输入，另一部分专门用于输出的缓冲结构。缓冲池则是把多个缓冲区连接起来统一管理，既可用于输入又可用于输出的缓冲结构。

显然，无论是多缓冲还是缓冲池，由于缓冲器是临界资源，在使用缓冲区时都有一个申请、释放和互斥的问题。下面以缓冲池为例介绍缓冲的管理。

### 9.4.3　缓冲池的管理

**1. 缓冲池的结构**

为了讨论缓冲池的管理，先来看看缓冲池的组成。缓冲池由多个缓冲区组成。而一个缓冲区由两部分组成：一部分是用来标识该缓冲器和用于管理的缓冲首部，另一部分是用于存放数据的缓冲体。这两部分有一一对应的映射关系。对缓冲池的管理是通过对每一个缓冲器的缓冲首部进行操作实现的。

缓冲首部如图 9.9 所示，它包括设备号、设备上的数据块号（块设备时）、互斥标识位以及缓冲队列连接指针和缓冲器号等。

系统把各缓冲区按其使用状况连成 3 种队列：

（1）空白缓冲队列 em，其队首指针为 F(em)，队尾指针为 L(em)。

（2）装满输入数据的输入缓冲队列 in，其队首指针为 F(in)，队尾指针为 L(in)。

（3）装满输出数据的输出缓冲队列 out，其队首指针为 F(out)，队尾指针为 L(out)。

其队列构成如图 9.10 所示。

| 设备号 |
|---|
| 数据块号 |
| 缓冲器号 |
| 互斥标识位 |
| 连接指针 |

图 9.9　缓冲首部

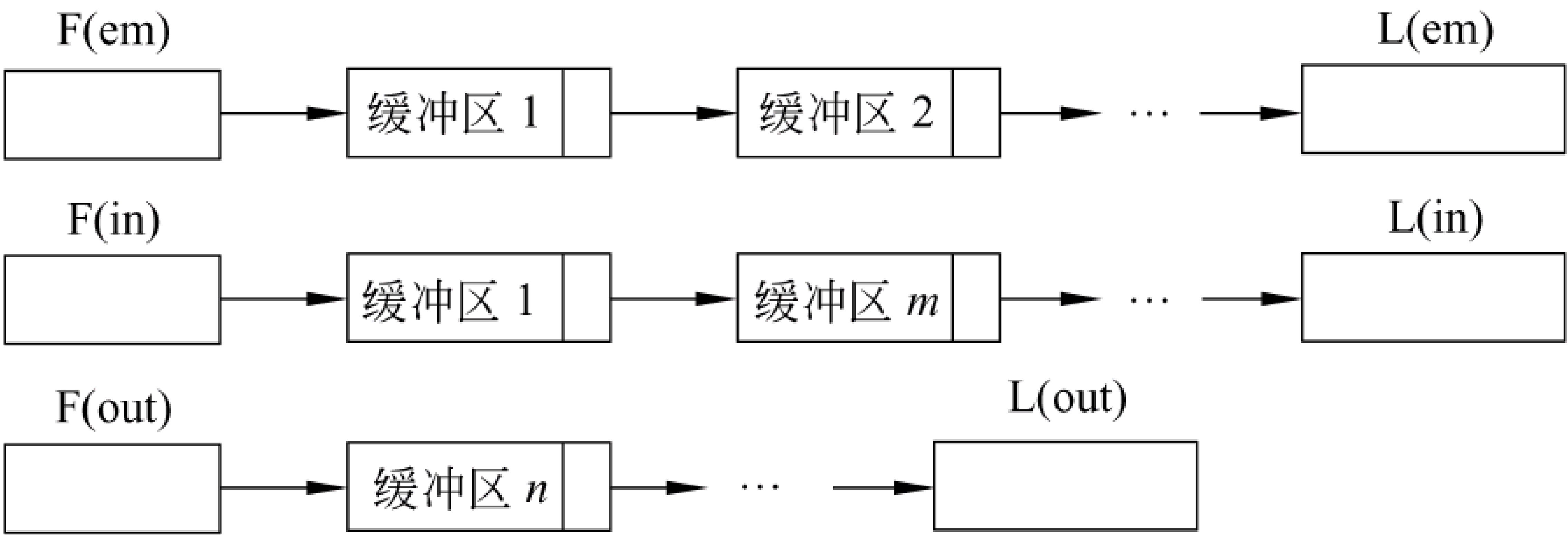



图 9.10　缓冲区队列

除了 3 种缓冲队列之外，系统（或用户进程）从这 3 种队列中申请和取出缓冲区，并用得到的缓冲区进行存数、取数操作，在存数、取数操作结束后，再将缓冲区放入相应的队列。这些缓冲区被称为工作缓冲区。在缓冲池中，有 4 种工作缓冲区：

（1）用于收容设备输入数据的收容输入缓冲区 hin；

（2）用于提取设备输入数据的提取输入缓冲区 sin；

(3) 用于收容 CPU 输出数据的收容输出缓冲区 hout；

(4) 用于提取 CPU 输出数据的提取输出缓冲区 sout。

缓冲池的工作缓冲区如图 9.11 所示。

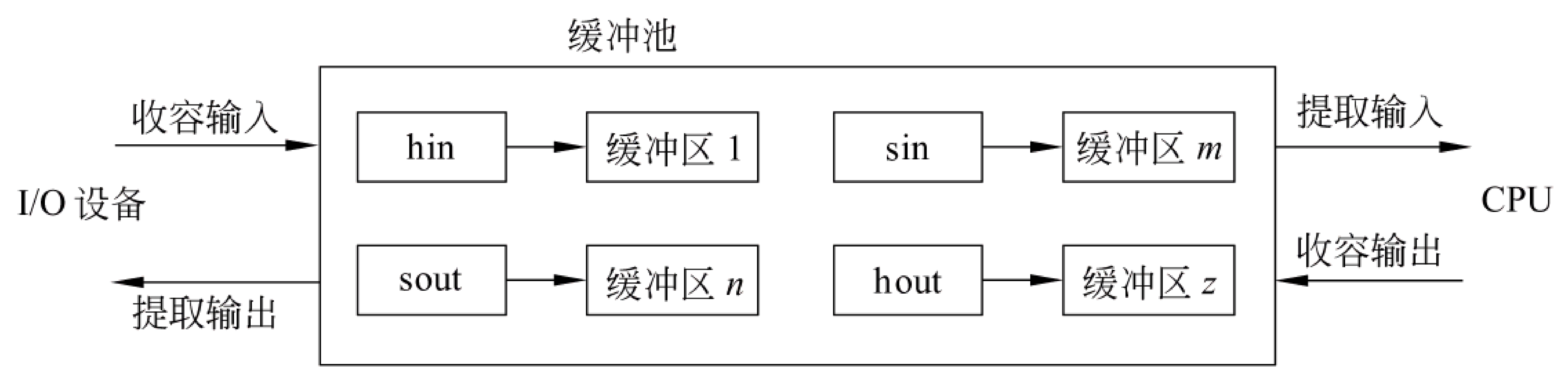



图 9.11　缓冲池的工作缓冲区

**2. 缓冲池管理**

对缓冲池的管理由如下几个操作组成：

(1) 从 3 种缓冲区队列中按一定的选取规则取出一个缓冲区的过程 take_buf(type)；

(2) 把缓冲区按一定的选取规则插入相应的缓冲区队列的过程 add_buf(type, number)；

(3) 供进程申请缓冲区用的过程 get_buf(type,number)；

(4) 供进程将缓冲区放入相应缓冲区队列的过程 put_buf(type,work_buf)。

其中，参数 type 表示缓冲队列类型，number 为缓冲区号，而 work_buf 则表示工作缓冲区类型。

使用这几个操作，缓冲池的工作过程可描述如下：

首先，输入进程调用 get_buf(em,number)过程从空白缓冲区队列中取出一个缓冲号为 number 的空白缓冲区，将其作为收容输入缓冲区 hin，当 hin 中装满了由输入设备输入的数据之后，系统调用过程 put_buf(in,hin)将该缓冲区插入输入缓冲区队列 in 中。

另外，当进程需要输出数据时，输出进程经过缓冲管理程序调用过程 get_buf(em, number)从空白缓冲区队列中取出一个空白缓冲区 number 作为收容输出缓冲区 hout，待 hout 中装满输出数据之后，系统再调用过程 put_buf(out,hout)将该缓冲区插入输出缓冲区队列 out。

对缓冲区的输入数据和输出数据的提取也是由过程 get_buf 和 put_buf 实现的。get_buf(out,number)从输出缓冲区队列中取出装满输出数据的缓冲区 number，将其作为 sout。当 sout 中的数据输出完毕时，系统调用过程 put_buf(em,sout)将该缓冲区插入空白缓冲区队列。而 get_buf(in,number)则从输入缓冲区队列中取出一个装满输入数据的缓冲区 number 作为输入缓冲区 sin，当 CPU 从中提取完所需数据之后，系统调用过程 put_buf(em,sin)将该缓冲区释放和插入空白缓冲区队列 em 中。

显然，对于各缓冲区队列中缓冲区的排列以及每次取出和插入缓冲区队列的顺序都应有一定的规则。最简单的方法是 FIFO(先进先出)的排列方法。采用 FIFO 方法，过程 put_buf 每次把缓冲区插入相应的缓冲区队列的队尾，而过程 get_buf 则取出相应的缓冲区队列的第一个缓冲区，从而 get_buf 中的第二个参数 number 可以省略。而且，采用 FIFO 方法也省略了对缓冲区队列的搜索时间。

过程 add_buf(type,number)和 take_buf(type,number)分别用来把缓冲区 number 插

入 type 队列和从 type 队列中取出缓冲区 number。它们分别被过程 get_buf 和 put_buf 调用,其中,take_buf 返回所取缓冲区 number 的指针,而 add_buf 则将给定缓冲区 number 的指针链入队列。

下面给出过程 get_buf 和 put_buf 的描述。

首先,设互斥信号量 S(type),其初值为 1。设描述资源数目的信号量 RS(type),其初值为 $n$($n$ 为 type 队列长度)。

```
get_buf(type,number):
    begin
        P(RS(type))
        P(S(type))
        Pointer of buffer(number)=take_buf(type,number)
        V(S(type))
    end
put_buf(type,number):
    begin
        P(S(type))
        add_buf(type,number)
        V(S(type))
        V(RS(type))
    end
```

## 9.5 设备分配

前面已经介绍了 I/O 数据传送控制方式及与其紧密相关的中断技术与缓冲技术。不过,在讨论这些问题时,已经做了如下假定:每一个准备传送数据的进程都已申请到了它所需要的外围设备、控制器和通道。事实上,由于设备、控制器和通道资源的有限性,不是每一个进程随时随地都能得到这些资源。进程必须首先向设备管理程序提出资源申请,然后,由设备分配程序根据相应的分配算法为进程分配资源。如果申请进程得不到它所申请的资源时,将被放入资源等待队列中等待,直到所需要的资源被释放。

下面,讨论设备分配和管理的数据结构、分配策略原则以及分配算法等。

### 9.5.1 设备分配用数据结构

设备的分配和管理通过下列数据结构进行。

**1. 设备控制表(Device Control Table,DCT)**

设备控制表(DCT)反映设备的特性、设备和 I/O 控制器的连接情况。包括设备标识、使用状态和等待使用该设备的进程队列等。系统中每个设备都必须有一张 DCT,且在系统中生成时或在该设备和系统连接时创建,但表中的内容则根据系统执行情况而被动态地修改。DCT 包括以下内容:

(1) 设备标识符,用来区别设备。

(2) 设备类型,反映设备的特性,例如是终端设备、块设备或字符设备等。

(3) 设备地址或设备号，由计算机原理课可知，每个设备都有相应的地址或设备号。这个地址既可以是和内存统一编址的，也可以是单独编址的。

(4) 设备状态，指设备是处于工作中还是空闲中。

(5) 等待队列指针，等待使用该设备的进程组成等待队列，其队首和队尾指针存放在DCT中。

(6) I/O控制器指针，该指针指向该设备相连接的I/O控制器。

**2. 系统设备表(System Device Table，SDT)**

系统设备表(SDT)整个系统一张，它记录已被连接到系统中的所有物理设备的情况，并为每个物理设备设一个表项。SDT的每个表项包括的内容如下：

(1) DCT指针，该指针指向有关设备的设备控制表。

(2) 正在使用设备的进程标识。

(3) 设备类型和设备标识符，该项的意义与DCT中的相同。

SDT的主要意义在于反映系统中设备资源的状态，即系统中有多少设备，有多少是空闲的，而又有多少已分配给了哪些进程。

**3. 控制器表(COntroller Control Table，COCT)**

COCT也是每个控制器一张，它反映I/O控制器的使用状态以及和通道的连接情况等(在DMA方式时，该项是没有的)。

**4. 通道控制表(CHannel Control Table，CHCT)**

该表只在通道控制方式的系统中存在，也是每个通道一张。CHCT包括通道标识符、通道忙/闲标识、等待获得该通道的进程等待队列的队首指针与队尾指针等。

SDT、DCT、COCT及CHCT如图9.12所示。

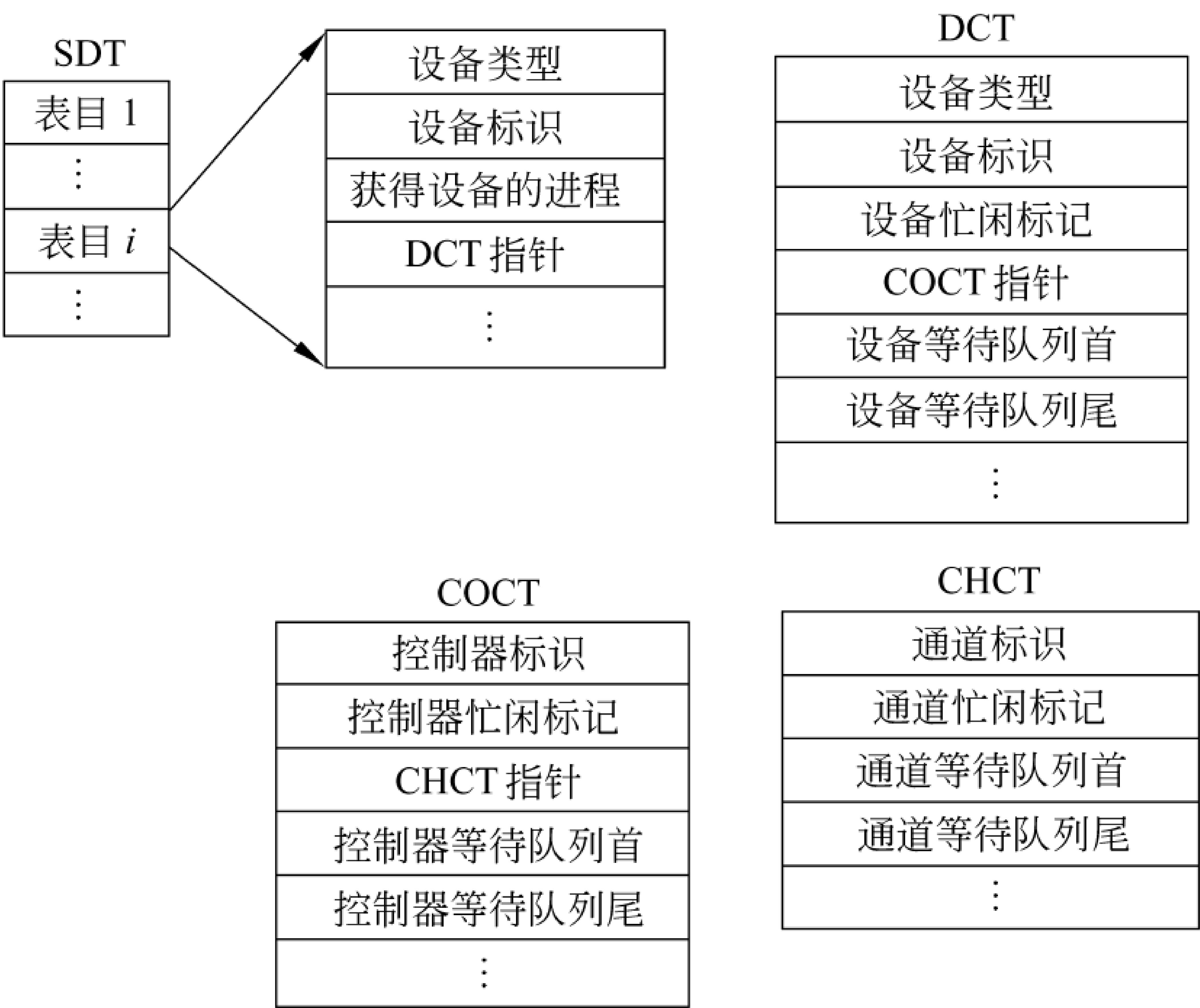



图9.12　数据结构表

显然，一个进程只有获得了通道、控制器和所需设备三者之后，才具备了进行I/O操作的物理条件。

## 9.5.2 设备分配的原则

### 1. 设备分配原则

设备分配的原则是根据设备特性、用户要求和系统配置情况决定的。设备分配的总原则是既要充分发挥设备的使用效率，尽可能地让设备忙，但又要避免由于不合理的分配方法造成进程死锁；另外还要做到把用户程序和具体物理设备隔离开来，即用户程序面对的是逻辑设备，而分配程序将在系统把逻辑设备转换成物理设备之后，再根据要求的物理设备号进行分配，如图 9.13 所示。

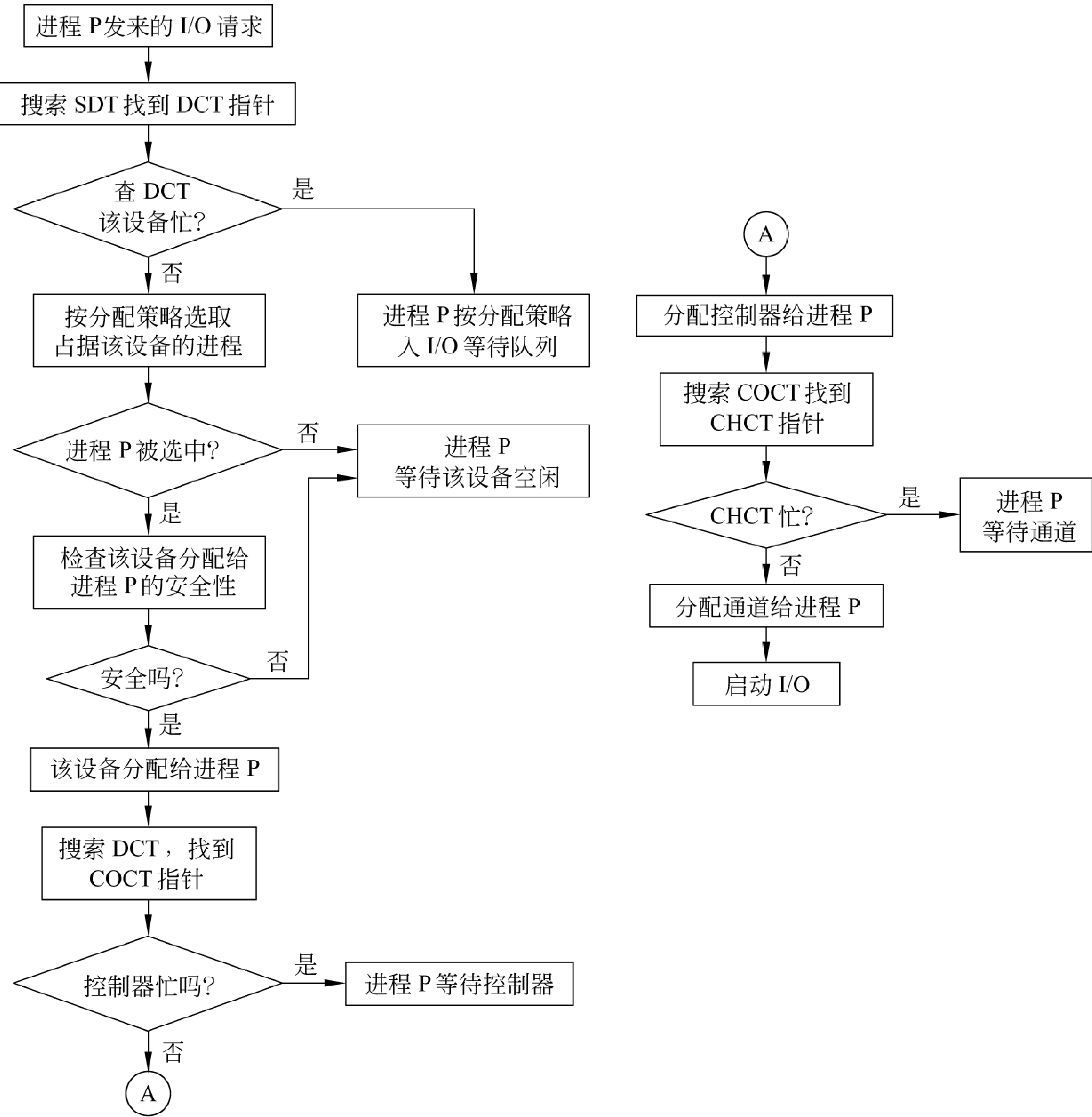



图 9.13 设备分配流程图

设备分配方式有两种，即静态分配和动态分配。静态分配方式是在用户作业开始执行之前，由系统一次分配该作业所要求的全部设备、控制器和通道。一旦分配之后，这些设备、控制器和通道就一直为该作业所占用，直到该作业被撤销。静态分配方式不会出现死锁，但设备的使用效率低。因此，静态分配方式并不符合设备分配的总原则。

动态分配在进程执行过程中根据执行需要进行。当进程需要设备时，通过系统调用命令向系统提出设备请求，由系统按照事先规定的策略给进程分配所需要的设备、I/O 控制器和通道，一旦用完之后，便立即释放。动态分配方式有利于提高设备的利用率，但如果分配算法使用不当，则有可能造成进程死锁。

**2. 设备分配策略**

与进程调度相似，动态设备分配也是基于一定的分配策略的。常用的分配策略有先请求先分配、优先级高者先分配策略等。

1）先请求先分配

当有多个进程对某一设备提出 I/O 请求时，或者是在同一设备上进行多次 I/O 操作时，系统按提出 I/O 请求的先后顺序，将进程发出的 I/O 请求命令排成队列，其队首指向被请求设备的 DCT。当该设备空闲时，系统从该设备的请求队列的队首取下一个 I/O 请求消息，将设备分配给发出这个请求消息的进程。

2）优先级高者先分配

优先级高者指发出 I/O 请求命令的进程。这种策略和进程调度的优先数法是一致的，即进程的优先级高，它的 I/O 请求也优先予以满足。对于相同优先级的进程来说，则按先请求先分配策略分配。因此，优先级高者先分配策略把请求某设备的 I/O 请求命令按进程的优先级组成队列，从而保证在该设备空闲时，系统能从 I/O 请求队列队首取下一个具有最高优先级进程发来的 I/O 请求命令，并将设备分配给发出该命令的进程。

### 9.5.3 设备分配算法

根据设备分配策略和原则，使用系统提供的 SDT、DCT、COCT 及 CHCT 等数据结构，当某个进程提出 I/O 设备请求之后，就可按图 9.13 所示的流程进行设备分配。

## 9.6 I/O 进程控制

### 9.6.1 I/O 控制的引入

前面各节在描述了 I/O 数据传送控制方式的基础上，讨论了中断、缓冲技术以及进行 I/O 数据传送所必需的设备分配策略与算法。那么，系统在何时分配设备，在何时申请缓冲，由哪个进程进行中断响应呢？另外，尽管 CPU 向设备或通道发出了启动指令，设备的启动以及 I/O 控制器中有关寄存器的值由谁来设置呢？这些都是前面几节的讨论中没有解决的问题。

从用户进程的输入输出请求开始，给用户进程分配设备和启动有关设备进行 I/O 操作，以及在 I/O 操作完成之后响应中断，进行善后处理为止的整个系统控制过程称为 I/O 控制。

### 9.6.2 I/O 控制的功能

I/O 控制的功能如图 9.14 所示。

I/O 控制过程首先收集和分析调用 I/O 控制过程的原因：是外设来的中断请求？还是

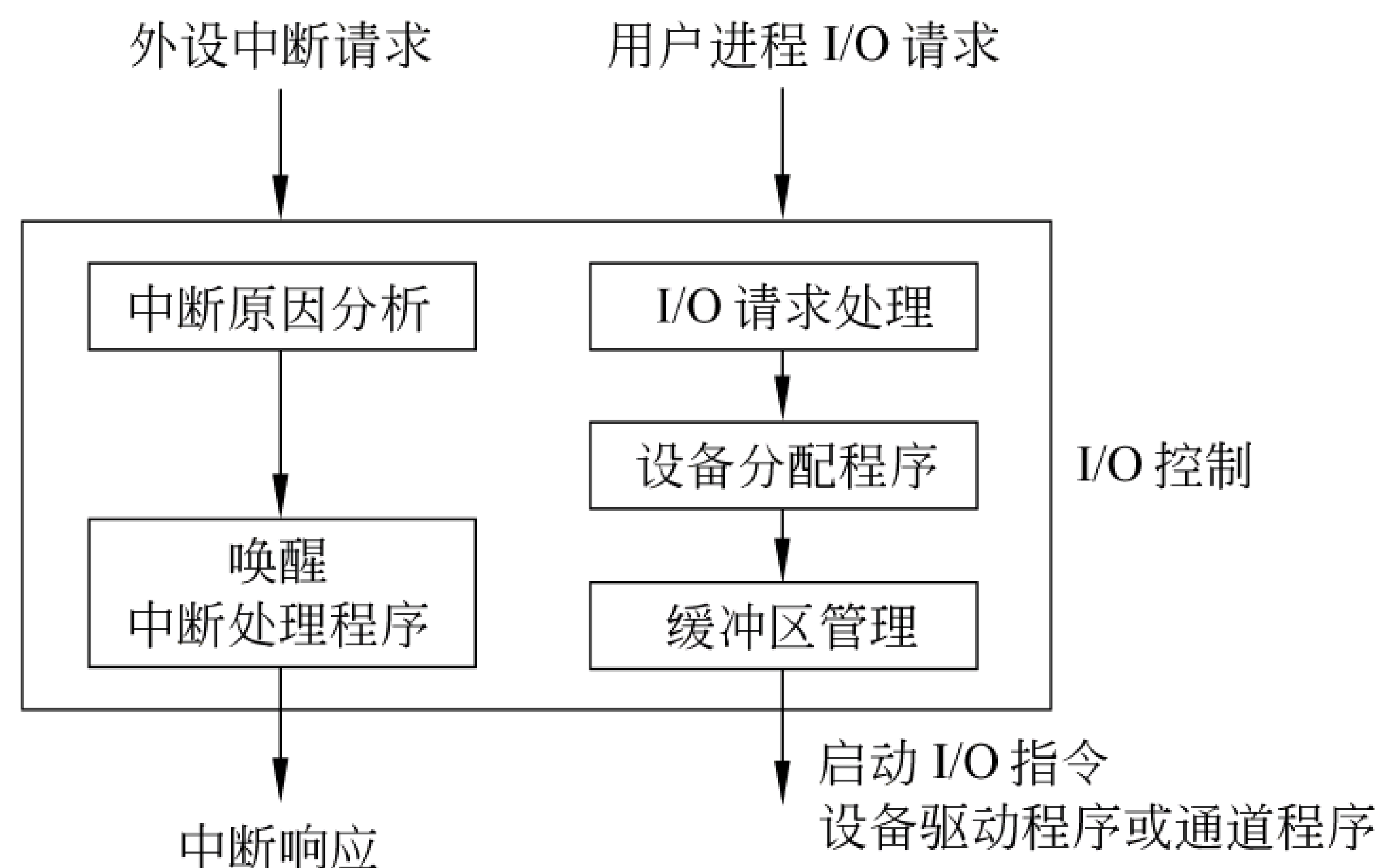



图 9.14 I/O 控制的功能

进程来的 I/O 请求？然后，根据不同的请求，分别调用不同的程序模块进行处理。

图 9.14 中各子模块的功能可简单地说明如下。

I/O 请求处理是用户进程和设备管理程序接口的一部分，它把用户进程的 I/O 请求变换为设备管理程序所能接收的信息。一般来说，用户的 I/O 请求包括所申请进行 I/O 操作的逻辑设备名、要求的操作、传送数据的长度和起始地址等。I/O 请求处理模块对用户的 I/O 请求进行处理。它首先将 I/O 请求中的逻辑设备名转换为对应的物理设备名；然后，检查 I/O 请求命令中是否有参数错误；在 I/O 请求命令参数正确时，它把该命令插入指向相应 DCT 的 I/O 请求队列；然后启动设备分配程序。在有通道的系统中，I/O 请求处理模块还将按 I/O 请求命令的要求编制出通道程序。

在设备分配程序为 I/O 请求分配了相应的设备、控制器和通道之后，I/O 控制模块还将启动缓冲区管理模块为此次 I/O 传送申请必要的缓冲区，以保证 I/O 传送的顺利完成。缓冲区的申请也可在设备分配之前进行。例如 UNIX 系统首先请求缓冲区，然后把 I/O 请求命令写到缓冲区中并将该缓冲区挂到设备的 I/O 请求队列上。

另外，在数据传送结束后，外设发出中断请求，I/O 控制过程将调用中断处理程序和做出中断响应。对于不同的中断，其善后处理也不同。例如处理结束中断时，要释放相应的设备、控制器和通道，并唤醒正在等待该操作完成的进程。另外，还要检查是否还有等待该设备的 I/O 请求命令。如有，则要通知 I/O 控制过程进行下一个 I/O 传送。

### 9.6.3 I/O 控制的实现

I/O 控制过程在系统中可以按 3 种方式实现：

(1) 作为请求 I/O 操作的进程的一部分实现。这种情况下，请求 I/O 操作的进程应具有良好的实时性，且系统应能根据中断信号的内容准确地调度到请求相应 I/O 操作的进程占据处理机，因为在大多数情况下，当一个进程发出 I/O 请求命令之后，都被阻塞进入睡眠态。

(2) 作为当前进程的一部分实现。这种情况下，不要求系统具有高的实时性。但由于当前进程与完成的 I/O 操作无关，所以当前进程不能接受 I/O 请求命令的启动 I/O 操作。不过，当前进程可以在接收到中断信号后，将中断信号转交给 I/O 控制模块处理，因此，如果让请求 I/O 操作的进程调用 I/O 操作控制部分(I/O 请求处理、设备分配、缓冲区分配等)，而让当前进程负责调用中断处理部分，也是一种可行的 I/O 控制方案。

(3) I/O 控制由专门的系统进程——I/O 进程完成。在用户进程发出 I/O 请求命令之后，系统调度 I/O 进程执行，控制 I/O 操作。同样，在外设发出中断请求之后，I/O 进程也被调度执行以响应中断。I/O 请求处理模块、设备分配模块以及缓冲区管理模块、中断原因分析模块、中断处理程序模块和后述的设备驱动程序模块等都是 I/O 进程的一部分。

I/O 进程也可分为 3 种方式实现：

(1) 每类(个)设备设一个专门的 I/O 进程，且该进程只能在系统态下执行。

(2) 整个系统设一个 I/O 进程，全面负责系统的数据传送工作。由于现代计算机系统设备十分复杂，I/O 进程的负担很重，因此，又可把 I/O 进程分为输入进程和输出进程。

(3) 每类(个)设备设一个专门的 I/O 进程，但该进程既可在用户态也可在系统态下执行。

## 9.7　设备驱动程序

设备驱动程序是驱动物理设备和 DMA 控制器或 I/O 控制器等直接进行 I/O 操作的子程序的集合。它负责设置相应设备有关寄存器的值，启动设备进行 I/O 操作，指定操作的类型和数据流向等。

为了对设备驱动程序进行管理，系统中设置有设备开关表(Device Switch Table，DST)。设备开关表中给出相应设备的各种操作子程序，例如打开、关闭、读、写和启动设备子程序的入口地址。一般来说，设备开关表是二维结构，其中的行和列分别表示设备类型和驱动程序类型。设备开关表也是 I/O 进程的一个数据结构。I/O 控制过程为进程分配设备和缓冲区之后，可以使用设备开关表调用所需的设备驱动程序进行 I/O 操作。

## 本 章 小 结

设备管理的主要任务是控制设备和 CPU 之间进行 I/O 操作。由于现代操作系统的外部设备的多样性和复杂性以及不同的设备需要不同的设备处理程序，设备管理成了操作系统中最复杂、最具有多样性的部分。设备管理模块在控制各类设备和 CPU 进行 I/O 操作的同时，还要尽可能地提高设备和设备之间、设备和 CPU 之间的并行操作度以及设备利用率，从而使得整个系统获得最佳效率。另外，设备管理模块还应该为用户提供一个透明的、易于扩展的接口，以使得用户不必了解具体设备的物理特性和便于设备的追加和更新。

围绕着上述目的，本章从设备的分类出发，对设备和 CPU 之间的数据传送的控制方式、中断和缓冲技术、设备分配原则和算法、I/O 控制过程以及设备驱动程序进行了介绍和讨论。

常用的设备和 CPU 之间的数据传送控制方式有 4 种，它们是程序直接控制方式、中断控制方式、DMA 方式和通道方式。程序直接控制方式和中断控制方式都只适用于简单的、外设很少的计算机系统，因为程序直接控制方式耗费大量的 CPU 时间，而且无法检测发现设备或其他硬件产生的错误，设备和 CPU、设备和设备只能串行工作。中断控制方式虽然在某种程度上解决了上述问题，但由于中断次数多，因而 CPU 仍需要花较多的时间处理中

断,而且能够并行操作的设备台数也受到中断处理时间的限制,中断次数增多导致数据丢失。DMA 方式和通道方式较好地解决了上述问题。这两种方式采用了外设和内存直接交换数据的方式。只有在一段数据传送结束时,这两种方式才发出中断信号要求 CPU 做善后处理,从而大大减少了 CPU 的工作负担。DMA 方式与通道控制方式的区别是,DMA 方式要求 CPU 执行设备驱动程序启动设备,给出存放数据的内存始址以及操作方式和传送字节长度等;而通道控制方式则是在 CPU 发出 I/O 启动命令之后,由通道指令来完成这些工作。

中断及其处理是设备管理中的一个重要部分。本章在介绍中断的基本概念的同时,对陷阱和软中断也做了相应的介绍和比较。另外,还介绍和描述了中断处理的基本过程。

缓冲是为了匹配设备和 CPU 的处理速度,以及为了进一步减少中断次数和解决 DMA 方式或通道方式时的瓶颈问题引入的。缓冲有硬缓冲和软缓冲之分。本章还介绍了对缓冲池的管理和操作。由于缓冲区是临界资源,所以对缓冲区或缓冲队列的操作必须互斥。

然后,介绍了设备分配的原则和算法。设备分配应保证设备有高的利用率和避免产生死锁。进程只有在得到了设备、I/O 控制器和通道(通道控制方式时)之后,才能进行 I/O 操作。另外,用户进程给出的 I/O 请求中包含逻辑设备号,设备管理程序必须将其变换成实际的物理设备。I/O 请求命令中的其他参数将被用来编制通道指令程序或由设备开关表选择设备驱动程序。

I/O 控制过程是对整个 I/O 操作的控制,包括对用户进程 I/O 请求命令的处理,进行设备分配和缓冲区分配,启动通道指令程序或驱动程序进行真正的 I/O 操作,以及分析中断原因和响应中断等。

## 习　　题

9.1　设备管理的目标和功能是什么?

9.2　数据传送控制方式有哪几种?试比较它们各自的优缺点。

9.3　什么是通道?试画出通道控制方式时的 CPU、通道和设备的工作流程图。

9.4　什么叫中断?什么叫中断处理?什么叫中断响应?

9.5　什么叫关中断?什么叫开中断?什么叫中断屏蔽?

9.6　什么是陷阱?什么是软中断?试述中断、陷阱和软中断之间的异同。

9.7　描述中断控制方式时的 CPU 动作过程。

9.8　什么是缓冲?为什么要引入缓冲?

9.9　设在对缓冲队列 em、in 和 out 进行管理时,采用最近最少使用算法存取缓冲区,即在一个缓冲区分配给进程之后,只要不是所有其他的缓冲区都在更近的时间内被使用过了,则该缓冲区不再分配出去。试描述过程 take_buf(type,number)和 add_buf(type,number)。

9.10　试述对缓冲队列 em、in 和 out 采用最近最少使用算法对改善 I/O 操作性能有什么好处。

9.11　用于设备分配的数据结构有哪些?它们之间的关系是什么?

9.12 设计一个设备分配的安全检查程序，以保证把某台设备分配给某进程时不会出现死锁。

9.13 什么是 I/O 控制？它的主要任务是什么？

9.14 I/O 控制可用哪几种方式实现？各有什么优缺点？

9.15 设备驱动程序是什么？为什么要有设备驱动程序？用户进程怎样使用设备驱动程序？

# 第 10 章　Linux 文件系统

## 10.1　Linux 文件系统的特点与文件类别

### 10.1.1　特点

在第 8 章中介绍了文件系统的基本原理。本章通过 Linux 的文件系统来进一步深入了解文件系统与操作系统其他部分的关系以及文件系统的设计方法。一个 Linux 的初始用户最先接受的概念可能是文件，因为用户首先必须把自己所输入的数据写入文件或启动有关系统文件执行。

在 Linux 系统中，所有的文件被组织到一个统一的树形目录结构中。也就是说，整个文件系统有一个“根”(/)，然后在根上分“杈”(目录)，任何一个分杈上都可以再分杈，杈上也可以长出“叶子”(文件)。“根”和“杈”在 Linux 中被称为 “目录”或“文件夹”。而“叶子”则是一个个的文件。这样不论底层存储设备是什么，展现在用户面前的均是一个统一的文件系统视图。

在 Linux 系统中，典型的目录结构如下：

| | |
|---|---|
| / | 根目录 |
| /bin | 存放常用的用户命令 |
| /boot | 存放内核及系统启动所需的文件 |
| /dev | 存放设备文件 |
| /etc | 存放配置文件 |
| /home | 用户文件的主目录 |
| /lib | 存放运行库 |
| /media | 其他文件系统的挂载点 |
| /mnt | 与/media 目录相同 |
| /proc | proc 文件系统的挂载点，用于存放进程和系统信息 |
| /root | 超级用户(root)的主目录 |
| /sbin | 存放系统管理程序 |
| /sys | 用于存放与设备相关的系统信息 |
| /tmp | 存放临时文件 |
| /usr | 存放应用软件包的主目录 |
| /usr/X11R6 | 存放 X Window 程序 |
| /usr/bin | 存放应用程序 |
| /usr/doc | 存放应用程序文档 |
| /usr/etc | 存放配置文件 |

/usr/games　存放游戏
/usr/include　存放 C 语言开发工具的头文件
/usr/lib　存放运行库
/usr/local　存放本地增加的应用程序
/usr/man　存放用户帮助文件
/usr/sbin　存放系统管理程序
/usr/share　存放结构独立的数据
/usr/src　存放程序源代码
/var　存放系统产生的文件，如日志等

以上列出的是一个典型的 Linux 系统目录结构。各个目录节点之下都会有一些文件和子目录。并且，系统在创建每一个目录时，都会自动为它设置两个目录文件，一个是“.”，代表该目录自己，另一个是“..”，代表该目录的父目录。对于根目录，“.”和“..”都代表其自己。

文件系统被组织成树形结构之后，文件名由路径名给出。路径名确定一个文件在文件系统中的位置。一个完整的路径名由代表根目录的斜杠开始，到所指定的文件为止。例如/usr/bin/man确定了文件 man 在文件系统中的位置。另外，路径名也可从正在执行进程的当前目录开始指定，例如，若当前目录是/home/zhang 的话，路径名 a.txt 与/home/zhang/a.txt 具有相同的效果。

一般来说，除了具有树形结构的特点之外，Linux 文件系统还具有如下特点：

- 文件是无结构的字符流式文件。
- 文件可以动态地增长或减少。
- 文件数据可由文件拥有者设置相应的访问权限而受到保护。
- 外部设备，例如磁盘设备、键盘、鼠标、串口等都被看作文件。从而，设备可通过文件系统隐蔽掉设备特性。在文件系统中，设备文件占据着文件系统目录结构中相应的位置，用户程序使用相同的系统调用和语法来读、写设备文件和普通文件。因此，用户程序既没有必要知道设备的内部特性，也不必在更换或增加设备之后修改自己。

相对于 Linux 2.4 内核版本，Linux 2.6 内核版本对文件系统做了一些改进，从本地文件系统看，Linux 2.6 内核支持日志文件系统功能，支持文件的扩展属性及 POSIX 标准访问控制；从网络文件系统来看，Linux 2.6 内核实验性地支持 NFSv4 版本在客户端和服务器端的实现，使网络文件系统管理更便捷，并加强了对 Windows 类型网络文件系统的支持。

### 10.1.2　文件类型

Linux 文件可分为 6 种类型，它们是普通文件、目录文件、设备文件（包括字符设备文件和块设备文件）、有名管道（FIFO）、软链接和 UNIX 域套接字。其中最常见的是普通文件、目录文件和设备文件 3 类。

普通文件即存储用户和系统的有关数据和程序的文件。它是无结构、无记录概念的字符流式文件。

目录文件则是由文件系统中的各个目录所形成的文件。这种文件在形式上同普通文件一样，由系统将其解释成目录。在 Linux 系统中，一个目录文件由多个目录项组成，而每个

目录项则由文件名及指示相应的文件索引节点(inode)的标识符 id 组成。

普通文件和目录文件都是无结构、无记录概念的字符流式文件。

设备文件与普通文件和目录文件不同,它除了在目录文件和文件索引节点表中占据相应的位置之外,并不占有实际的物理存储块。因此,对设备文件的读、写操作将实际上变为对设备的操作,而对设备文件的保护也将变成对设备的保护。例如:

```
# cp /dev/tty1 terminalread
```

把在第一个虚拟终端上输入的字符(设备文件/dev/tty1 是用户虚拟终端 1)读入,并把它们复制到文件 terminalread 上。

# 10.2 Linux 的虚拟文件系统

## 10.2.1 虚拟文件系统框架

随着需求的增长,要求操作系统能够支持多种不同的文件系统。为实现这个功能,Linux 内核使用了虚拟文件系统。(Virtual File System,或称为 Virtual Filesystem Switch,VFS)。VFS 是 Linux 内核中的一个软件层,用于给用户空间的程序提供文件系统接口。它也提供了内核中的一个抽象功能,允许不同的文件系统共存。VFS 隐藏了各种硬件的具体细节,为所有的文件系统操作提供了统一的接口。这样,借助 VFS,在 Linux 系统中可以使用多个不同的文件系统。不同的文件系统被挂装以后,对它们的使用与传统的单一文件系统没有区别。

VFS 是由面向对象的思想发展起来的,VFS 提供一个抽象基类,由这个基类派生出的子类支持具体的文件系统。VFS 只代表内核中的一个文件系统,而 VFS 的索引节点 inode 只代表内核中的一个文件,它们只存在于内核中。真正的文件系统,如 ext2、nfs 和 vfat 等,必须在 VFS 提供的统一的接口支持下才能工作。VFS 提供的公共接口对于应用程序而言是透明的,当应用程序进行文件系统操作时,内核的文件子系统将首先调用 VFS 的相应函数,该函数先处理与设备无关的操作,然后根据 VFS 结构以及它的 inode 节点提供的信息,调用真正的文件系统中对应的函数,处理与设备相关的操作。

Linux 的虚拟文件系统为多种类型的文件系统提供了支持,包括:

(1) 基于磁盘的文件系统,如 ext2、ext3、reiserfs、JFS 和 XFS 等;以及 UNIX System Ⅴ 的文件系统;微软公司开发的 MS_DOS、vfat 及 ntfs;ISO 9660 光盘文件系统等。

(2) 基于网络的文件系统,如 NFS、SMB 和 OCFS 等。

(3) 特殊的文件系统,如 proc 和 sysfs。它们并不管理真正的磁盘空间,而是通过它们访问内核数据。

## 10.2.2 Linux 虚拟文件系统的数据结构

虚拟文件系统(VFS)在文件系统中引入了一个通用文件模型,这个通用文件模型是面向对象的。由于 Linux 没有使用面向对象的程序设计语言开发,而是使用 C 语言开发,因此对象是用数据结构实现的。这个模型由下列主要对象组成:

- 超级块(superblock)。存放已挂装文件系统的有关信息。

- 索引节点(inode)。存放关于一个具体文件的一般信息。每个索引节点分配一个索引节点号,用来指示文件系统中的指定文件。
- 文件(file)。存放打开文件与进程之间进行交互的有关信息。
- 目录项(dentry)。保存目录项与相应文件进行链接的信息。

**1. VFS 的超级块**

内核为每一个已挂装的文件系统分配一个超级块,所有的超级块对象组成一个链表。超级块数据结构在下面列出,为节省篇幅,只列出主要成员:

```
struct super_block {
    struct list_head            s_list;            /* 指向超级块链表的指针 */
    dev_t                       s_dev;             /* 设备标识符 */
    unsigned long               s_blocksize;       /* 以字节为单位的块大小 */
    unsigned char               s_blocksize_bits;  /* 以位为单位的块大小 */
    unsigned char               s_dirt;            /* 修改(脏)标志 */
    unsigned long long          s_maxbytes;        /* 文件最大尺寸 */
    struct file_system_type *   s_type;            /* 文件系统类型 */
    struct super_operations *   s_op;              /* 超级块方法 */
    struct dquot_operations *   dq_op;             /* 磁盘限额方法 */
    struct quotactl_ops *       s_qcop;            /* 磁盘限额管理方法 */
    struct export_operations *  s_export_op;       /* NFS 的输出方法 */
    unsigned long               s_flags;           /* 挂装标志 */
    unsigned long               s_magic;           /* 文件系统的魔数 */
    struct dentry          * s_root;      /* 文件系统挂装目录的目录项结构 */
    struct rw_semaphore    s_umount;      /* umount 使用的信号 */
    struct mutex           s_lock;        /* 锁标志 */
    int                    s_count;       /* 参考计数器 */
    …
    struct list_head       s_inodes;      /* 所有索引节点的链表 */
    struct list_head       s_dirty;       /* 所有已修改(脏)的索引节点的链表 */
    struct list_head       s_io;          /* 所有等待写回磁盘的索引节点链表 */
    struct hlist_head      s_anon;        /* 用于处理 NFS 输出的匿名目录项链表 */
    struct list_head       s_files;       /* 所有文件对象链表 */

    struct block_device    * s_bdev;      /* 指向块设备驱动描述符的指针 */
    …
    char                   s_id[32];      /* 包含这个超级块的块设备名 */
    void                   * s_fs_info;   /* 指向特定文件系统超级块信息数据结构的指针 */
    …
}
```

所有的超级块对象用双向循环链表的方式链接在一起。链表的第一个和最后一个元素分别存放在 super_blocks 变量的 slist 域的 next 和 prev 域中。数据结构 struct list_head 仅仅包括指向链表的前一个和后一个元素的指针。

域 s_fs_info 是一个指针,指向具体文件系统的超级块信息数据结构。对于 ext2 文件

系统，这个数据结构是 struct ext2_sb_info。其中包括了磁盘片大小、每个块组中的块数和磁盘分配位屏蔽等与 VFS 的通用文件模型无关的数据。在挂装磁盘文件系统时，内核从磁盘的超级块中把相应的数据读入内存的超级块信息数据结构中，然后就对内存中的数据进行操作，来提高效率。

与超级块关联的方法就是超级块操作，这些操作是由数据结构 struct super_operations 描述的，其地址存放在超级块的 s_op 域中。

```
struct super_operations {
    struct inode * (* alloc_inode)(struct super_block * sb);
    void (* destroy_inode)(struct inode * );
    void (* read_inode) (struct inode * );
    void (* dirty_inode) (struct inode * );
    int (* write_inode) (struct inode * , int);
    void (* put_inode) (struct inode * );
    void (* drop_inode) (struct inode * );
    void (* delete_inode) (struct inode * );
    void (* put_super) (struct super_block * );
    void (* write_super) (struct super_block * );
    int (* sync_fs)(struct super_block * sb, int wait);
    void (* write_super_lockfs) (struct super_block * );
    void (* unlockfs) (struct super_block * );
    int (* statfs) (struct super_block * , struct kstatfs * );
    int (* remount_fs) (struct super_block * , int * , char * );
    void (* clear_inode) (struct inode * );
    void (* umount_begin) (struct super_block * );
    int (* show_options)(struct seq_file * , struct vfsmount * );
    ssize_t (* quota_read)(struct super_block * , int, char * , size_t, loff_t);
    ssize_t (* quota_write)(struct super_block * , int, const char * , size_t,loff_t);
};
```

每个具体的文件系统都应该提供这些超级块操作的具体实现。这些超级块操作可以实现文件系统的挂装、卸载和读写 inode 节点等。

**2. 索引节点**

文件系统处理文件所需要的所有信息都存放在索引节点数据结构中。下面列出 struct inode 的主要成员：

```
struct inode {
    struct hlist_node      i_hash;       /* 指向散列链表的指针 */
    struct list_head       i_list;       /* 指向描述索引节点当前状态的链表的指针 */
    struct list_head       i_sb_list;    /* 指向超级块的索引节点链表的指针 */
    struct list_head       i_dentry;     /* 指向目录项链表的指针 */
    unsigned long          i_ino;        /* 索引节点号 */
    atomic_t               i_count;      /* 引用计数器 */
    umode_t                i_mode;       /* 文件类型与访问权限 */
    unsigned int           i_nlink;      /* 硬链接数目 */
```

```
    uid_t                       i_uid;          /* 所有者标识符 */
    gid_t                       i_gid;          /* 组标识符 */
    dev_t                       i_rdev;         /* 真实设备标识符 */
    loff_t                      i_size;         /* 文件的字节数 */
    struct timespec             i_atime;        /* 上次访问文件的时间 */
    struct timespec             i_mtime;        /* 上次修改文件的时间 */
    struct timespec             i_ctime;        /* 上次修改索引节点的时间 */
    unsigned int                i_blkbits;      /* 每块的位数 */
    unsigned long               i_blksize;      /* 每块的字节数 */
    unsigned long               i_version;      /* 版本号 */
    unsigned long               i_blocks;       /* 文件的块数 */
    unsigned short              i_bytes;        /* 文件占用的最后一块的字节数 */
    spinlock_t                  i_lock;         /* 保护索引节点的某些域的自旋锁 */
    struct mutex                i_mutex;        /* 互斥量 */
    struct rw_semaphore         i_alloc_sem;    /* 读写保护信号量,用于在直接 I/O 访 */
                                                /* 问文件时避免竞争关系 */
    struct inode_operations     *i_op;          /* 索引节点方法 */
    struct file_operations      *i_fop;         /* 默认的文件操作 */
    struct super_block          *i_sb;          /* 指向超级块对象的指针 */
    struct file_lock            *i_flock;       /* 指向文件锁链表的指针 */
    struct address_space        *i_mapping;     /* 指向地址空间对象的指针 */
    struct address_space        i_data;         /* 文件的地址空间对象 */
#ifdef   CONFIG_QUOTA
    struct dquot                *i_dquot[MAXQUOTAS];  /* 索引节点的磁盘限额 */
#endif
    …
    unsigned long               i_state;        /* 索引节点状态标志 */
    unsigned long               dirtied_when;   /* 索引节点修改时间 */
    unsigned int                i_flags;        /* 文件系统挂装标志 */
    atomic_t                    i_writecount;   /* 写进程的使用计数器 */
    void                        *i_security;    /* 指向索引节点安全结构的指针 */
    union {
        void                    *generic_ip;    /* 指向文件系统具体数据的指针 */
    } u;
    …
};
```

索引节点的域 i_count 是一个引用计数器。系统对 i 节点进行操作时,内核的 VFS 子系统中的操作把这个值加 1 或减 1。如果计数器值为 0,则可以把这个内存索引节点删除或分配给其他文件。

联合域 u 用于存放属于具体文件系统的索引节点信息。如果索引节点指向的是一个 ext2 文件,那么这个域就指向一个 ext2_inode_info 结构。

每个索引节点都要复制磁盘索引节点包含的一些数据,比如文件占用的磁盘块数等。如果索引节点的 i_state 域的值是 I_DIRTY_SYNC、I_DIRTY_DATASYNC 或 I_DIRTY_PAGES,表示该索引节点是“脏”的,说明该索引节点所对应的磁盘索引节点必须被更新。

根据索引节点的状态不同，每个 VFS 索引节点总会出现在下列循环双向链表的某个表中：

- 未使用的索引节点链表。这个表中的索引节点反映有效的磁盘索引节点，并且当前没有被任何进程使用。这些节点不是脏的，而且 i_count 域的值是 0。这个链表的第一个和最后一个元素的指针分别保存在 inode_unused 变量的 next 和 prev 域中。这个链表用作磁盘高速缓存。
- 正在使用的索引节点链表。这些索引节点反映有效的磁盘索引节点，并且当前正在被某些进程使用。这些索引节点不是“脏”的，而且 i_count 域的值为正数。链表的第一个和最后一个元素保存在 inode_in_use 变量中。
- 脏索引节点链表。由相应超级块对象的 s_dirty 域指向链表中的首尾元素。

这些链表都通过适当的索引节点列表的 i_list 域链接在一起。

此外，每个挂载的文件系统还有一个双向循环链表。每个索引节点都被包含在对应文件系统的链表中。文件系统超级块对象的 s_inodes 域指向该链表的首尾元素，索引节点对象的 i_sb_list 域保存指向链表中相邻元素的指针。

最后，索引节点对象还被保存在名为 inode_hashtable 的散列表中。散列表可以加速内核对索引节点对象的搜索，条件是内核知道索引节点号和相应文件所在的文件系统的超级块的地址。由于散列技术可能引起冲突，所以索引节点对象设置一个 i_hash 域，指向散列到同一地址的其他索引节点。

与索引节点相联系的方法叫索引节点操作，由 inode_operations 结构来描述，该结构的地址保存在 i_op 域中。该结构也包括一个指向文件操作方法的指针。

```
struct inode_operations {
    int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,int);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,int,dev_t);
    int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *,int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int, struct nameidata *);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
```

```
    void (* truncate_range) (struct inode * , loff_t, loff_t);
};
```

这些方法对所有的索引节点和文件系统类型都是可用的。

**3. 文件**

文件对象描述的是进程和一个打开文件交互的过程。文件对象是在文件被打开的时候创建的,由一个 file 结构组成。file 对象的主要成员如下。

```
struct file {
    …
    struct list_head            f_list;          /* 指向文件对象链表的指针 */
    struct dentry               * f_dentry;      /* 文件对应的目录项对象的指针 */
    struct vfsmount             * f_vfsmnt;      /* 包含这个文件的已挂装的文件系统 */
    struct file_operations      * f_op;          /* 指向文件操作表的指针 */
    atomic_t                    f_count;         /* 引用计数器 */
    unsigned int                f_flags;         /* 打开文件时指定的标志 */
    mode_t                      f_mode;          /* 进程的访问模式 */
    loff_t                      f_pos;           /* 当前位移量(文件指针) */
    struct fown_struct          f_owner;         /* 通过信号进行 I/O 事件通知的数据 */
    unsigned int                f_uid, f_gid;    /* 用户的 UID 和 GID */
    …
}
```

注意,文件对象在磁盘上没有相应的映像,因此 file 结构中没有“脏”域来表示文件对象是否被修改。

存放在文件对象中的主要信息是文件指针,即文件的当前位置,对文件的下一次操作从这里开始进行。由于几个进程可能并发访问同一个文件,因此文件指针不能存放在索引节点对象中。

文件对象存放在名为 filp 的 slab 分配器高速缓存中,缓存的描述地址保存在 filp_cachep 变量中。系统能分配的文件对象的总数是有限的,files_stat 变量的 max_files 域指明了能够分配的文件对象总数的最大值,也就是操作系统能同时打开的文件总数的最大值。

每个文件对象总包含在下列的一个双向循环链表中:

- 未使用的文件对象链表。链表的首元素存放在 free_list 变量中。
- 正在使用,但是没有分配给超级块的文件对象链表。链表的首元素存放在 anon_list 变量中。
- 正在使用,而且分配给超级块的文件对象链表。每个超级块对象把文件对象链表的首元素保存在它的 s_files 域中,这样属于不同文件系统的文件对象就包含在不同的链表中。

很明显,由文件对象构成的链表就是 Linux 操作系统的系统打开文件表。

文件对象的 f_count 域是一个参考计数器,记录正在使用这个文件对象的进程数。在父、子进程共享同一文件时,它们使用同一个文件对象。

当一个进程需要打开一个文件时,VFS 调用 get_empty_filp()函数,分配一个新的文件对象。该函数检测“未使用”链表的元素个数是否多于 NR_RESERVED_FILES,如果是,那

么新打开的文件可以使用其中的一个元素;否则,系统分配新的内存。

成员 f_op 指向一个文件操作表。每个文件系统都有自己的文件操作集合,执行诸如读写文件这样的操作。当内核将一个索引节点从磁盘装入内存时,就会把指向这些文件操作的指针存放在 file_operations 结构中,而该结构的地址存放在索引节点对象的 i_fop 域。当进程打开这个文件时,VFS 用这个地址初始化新文件对象的 f_op 域,使对文件操作的后续调用能使用这些函数。

**4. 目录项**

在 Linux 系统中,目录也是一类文件。当目录被读入内存,VFS 就把它转换为基于 dentry 结构的一个目录项对象。超级块 super_block 数据结构的 s_root 域记录了文件系统挂装目录的基本信息。进程查找路径名时,对路径名中包含的每个目录,VFS 都为其创建一个目录项对象。目录项对象将每个目录与其对应的索引节点相联系。

```
struct dentry {
    atomic_t        d_count;                    /* 目录项对象引用计数器 */
    unsigned        int d_flags;                /* 目录项标志 */
    spinlock_t      d_lock;                     /* 用于保护 dentry 对象的自旋锁 */
    struct inode    *d_inode;                   /* 与文件名关联的索引节点 */
    struct dentry   *d_parent;                  /* 父目录的目录项对象 */
    struct          qstr d_name;                /* 文件名 */
    struct          list_head d_lru;            /* 用于未使用链表的指针 */
    struct          list_head d_child;          /* 用于父目录的目录项对象的链表的指针 */
    struct          list_head d_subdirs;        /* 所有子目录的目录项链表 */
    …
    struct          dentry_operations d_op;     /* 目录项方法 */
    struct          super_block * d_sb;         /* 文件的超级块对象 */
    void            *d_fsdata;                  /* 依赖于文件系统的数据 */
    …
    struct          hlist_node d_hash;          /* 用于散列表表项的指针 */
    int             d_mounted;                  /* 挂装在这个目录项对象上的文件系统数目 */
    …
};
```

注意,目录项对象在磁盘上没有对应的映像,所以在 dentry 结构中不包含"脏"域。

由于从磁盘读入一个目录文件并构造相应的目录项对象要耗费大量的时间,所以,为了提高处理目录项对象的效率,Linux 使用目录项高速缓存(dentry cache)。对于所有仍然保存在目录项高速缓存中的目录项对象,与它相关的索引节点也会保存在内存的索引节点高速缓存(inode cache)中,而不会被释放。

### 10.2.3 VFS 的系统调用

VFS 是应用程序和具体文件系统之间的一层,它实现了与文件系统相关的所有系统调用,为各种文件系统提供一个通用的接口。应用程序不需要关心文件系统的实现细节,只需要与 VFS 进行交互。

例如,如果要实现如下一条 Shell 命令:

```
$cp /mnt/floppy/TEST /tmp/test
```

其中,/mnt/floppy/是软盘上的 MS-DOS 文件系统的挂装目录,而/tmp 是在 ext2 文件系统上的一个目录。然而 cp 程序不需要了解这些,它只需要使用标准的系统调用来实现。

```
inf=open("/mnt/floppy/TEST", O_RDONLY, 0);
outf=open("/tmp/test", O_WRONLY|O_CREATE|O_TRUNC, 0600);
do {
    l=read(inf, buf, 4096);
    write(outf, buf, l);
} while(l);
close(outf);
close(inf);
```

VFS 处理的一些系统调用如下:

| | |
|---|---|
| mount(), umount() | 挂装/卸载文件系统 |
| sysfs() | 获取文件系统信息 |
| statfs(), fstatfs(), ustat() | 获取文件系统统计信息 |
| chroot() | 更改根目录 |
| chdir(), fchdir(), getcwd() | 操纵当前目录 |
| mkdir(), rmdir() | 创建/删除目录 |
| stat(), fstat(), lstat(), access() | 读取文件状态 |
| open(), close(), creat(), umask() | 打开/关闭文件 |
| dup(), dup2(), fcntl() | 对文件描述符进行操作 |
| select(), poll() | 异步 I/O 通告 |
| read(), write(), readv(), writev(), sendfile() | 进行文件 I/O 操作 |
| readlink(), symlink() | 对软链接进行操作 |
| chown(), fchown(), lchown() | 更改文件所有者 |
| chmod(), fchmod(), utime() | 更改文件属性 |
| pipe() | 进行通信操作 |

对这些系统调用的具体说明,请参考 Linux 的用户手册。

## 10.3 文件系统的注册和挂装

内核通过 VFS 使用一个具体的文件系统之前,必须首先对这个文件系统进行注册和挂装。只有完成对某个文件系统的注册,内核才能使用这个具体文件系统的各种操作函数,将这个文件系统挂装的系统的目录树上。

### 10.3.1 文件系统注册

所谓注册,就是把某个具体文件系统的操作代码装入内核。对具体文件系统的操作代码或者被包含在内核映像中,或者作为一个模块,被动态装载。为了把对 VFS 超级块和索引节点的操作定向到对应的文件系统上,也就是实现从虚拟文件系统到实际文件系统的转

换,内核必须正确地对系统中所有文件系统进行跟踪和配置。

在用户空间中,可以从/proc/filesystems 文件读到所有已经在内核中注册的文件系统。在内核中,每个已经注册的文件系统用一个 file_system_type 数据结构描述。这个数据结构的主要成员如下:

```
struct file_system_type {
        const char * name;                              /* 文件系统名 */
        int fs_flags;                                   /* 文件系统类型标志 */
        struct super_block * (* get_sb) (struct file_system_type * , int, const
        char * , void * );                              /* 读超级块的方法 */
        void (* kill_sb) (struct super_block * );       /* 删除超级块的方法 */
        struct module * owner;                          /* 实现文件系统的模块 */
        struct file_system_type * next;                 /* 指向下一个链表元素的指针 */
        struct list_head fs_supers;                     /* 超级块对象链表的头 */
};
```

所有已经注册的文件系统类型都装入一个简单链表中,变量 file_systems 指向链表的第一个元素,而每个元素的 next 域指向链表的下一个元素。内核通过文件系统类型链表来搜索每个文件系统的接口函数,以便装入该文件的超级块,实现虚实文件系统的转换。

数据结构的成员 get_sb 指向与文件系统相关的函数,这个函数从磁盘设备读取超级块,并写入对应的 VFS 超级块对象。而 kill_sb 域指向的函数执行删除超级块的工作。

Linux 内核在编译时就已确定所支持的文件系统类型。这些文件系统在系统引导并初始化时,由内核调用函数 register_filesystem()进行注册,这个函数把相应的 file_system_type 对象插入文件系统类型链表中。

如果文件系统是一个内核可装载的模块,那么在模块被装入时,也要调用 register_filesystem()函数向文件系统类型注册链表进行注册。要注册的文件系统可能是 MS-DOS 文件系统,或者是 ISO 9660 文件系统,等等。

一种新的文件系统可以被注册,当然也可以被注销。在相应的模块被卸载时,调用 unregister_filesystem()函数,即从注册链表中删除该文件系统的数据结构。一旦被删除,系统将不再支持该种文件系统。

### 10.3.2 已挂装文件系统描述符链表

每个文件系统都有自己的挂装根目录。如果某个文件系统的根目录是系统文件树的根目录,那么该文件系统被称为根文件系统。而其他文件系统可以挂装到系统的目录树上,把这些文件系统要插入的目录称为挂装点(mount point)。新挂装的文件系统就是挂装点目录所在的文件系统的孩子。例如,对/文件系统和/proc 虚拟文件系统来说,/文件系统是父文件系统,/proc 文件系统是子文件系统。

对于每个操作系统,内核必须保存已经挂装文件系统的信息,包括挂装点和挂装标志,以及与其他已挂装文件系统之间的关系。这样的信息称为已挂装文件系统描述符,每个描述符是一个 vfsmount 类型的数据结构。

```
struct vfsmount
```

```
{
    struct list_head mnt_hash;           /* 用于散列链表的指针 */
    struct vfsmount *mnt_parent;         /* 父文件系统描述符。本文件系统挂装在其上 */
    struct dentry *mnt_mountpoint;       /* 指向这个文件系统挂装点的目录项 */
    struct dentry *mnt_root;             /* 指向这个文件系统根目录的目录项 */
    struct super_block *mnt_sb;          /* 指向这个文件系统的超级块 */
    struct list_head mnt_mounts;         /* 挂装在这个文件系统上的子文件系统的链表的头 */
    struct list_head mnt_child;          /* 父文件系统的 mnt_mount 链表所使用的指针 */
    atomic_t mnt_count;                  /* 引用计数器 */
    int mnt_flags;                       /* 标志 */
    char *mnt_devname;                   /* 设备文件名,如/dev/hda1 */
    struct list_head mnt_list;           /* 指向描述符全局链表的指针 */
    …
};
```

在内核中,vfsmount 数据结构保存在几个双向循环链接表中:

- 包含所有已安装文件系统描述符的双向循环全局链表,也被称为已挂装文件系统链表。这个链表的头保存在 vfsmntlist 变量中。描述符的 mt_list 字段包含链表中指向相邻元素的指针。
- vfsmount 描述符保存在一个散列表中,这个散列表由父文件系统的 vfsmount 描述符的地址和安装点目录的目录项对象的地址索引。散列表存放在 mount_hashtable 数组中,其大小依赖于系统中 RAM 的容量。描述符的 mnt_hash 用于这个散列表。
- 对于每一个安装的文件系统,所有安装在它之上的子文件系统形成了一个双向循环链表。每个链表的头存放在安装的文件系统描述符的 mnt_mounts 字段。子文件系统描述符的 mnt_child 字段存放指向链表中相邻元素的指针。

### 10.3.3 挂装根文件系统

挂装根文件系统是 Linux 操作系统初始化的一个关键部分。

当系统启动时,内核就要从变量 ROOT_DEV 中查找包含根文件系统的磁盘主设备号。这个变量或者在编译内核时指定,或者由引导程序向内核传递一个 root 参数。

挂装根文件系统分为两个阶段:

(1) 内核安装特殊的 rootfs 文件系统,该文件系统仅提供一个作为初始安装点的空目录。

(2) 内核在空目录上安装一个真正的根目录。

为什么内核要在安装实际根目录之前安装 rootfs 文件系统呢? 我们知道,rootfs 文件系统允许内核能够较容易地改变实际根文件系统。事实上,在某些情况下,内核需要一个接一个地安装和卸载几个根文件系统。例如,一个发布版的初始启动软盘可能把具有一组最小驱动程序的内核装入 RAM 中,内核把存放在 RAM 磁盘中的一个最小的文件系统作为根安装。接下来,在这个初始根文件系统中的程序探测系统的硬件,装入所有必需的内核模块,并从物理块设备重新安装文件系统。

第一阶段由 init_mount_tree()完成的,该函数在系统初始化过程中执行。它首先初始

化一个 file_system_type 数据结构，文件系统名 name 字段设为 rootfs。然后传递给 register_filesystem()函数，进行文件系统类型注册。然后调用 do_kern_mount()函数挂装这个特殊的文件系统，并返回新安装文件系统对象的地址；这个地址保存在 root_vfsmnt 变量中。从现在开始，root_vfsmnt 表示所安装文件系统树的根。

根文件系统安装操作的第二阶段是由 mount_root()函数在系统初始化即将结束时进行的。为了简单起见，只考虑基于磁盘文件系统的情况，在这种情况下，该函数执行下列主要操作：

(1) 检查 ROOT_DEV 根设备是否存在，是否正常工作。

(2) 扫描文件系统类型链表，对链表上的每个文件系统类型对象都调用相应的 read_super()，试图从 ROOT_DEV 磁盘设备读取超级块。由于每个文件系统的方法都有一个唯一的魔数，因此除了根文件系统对 read_super()的调用会成功之外，其他文件系统的调用都将失败。同时 read_super()函数还为根目录创建一个索引节点对象和一个目录项对象。

(3) 调用 add_vfsmnt，把第一个描述根文件系统的 vfsmount 结构插入到已挂装文件系统链表。

(4) 把 current(init 进程)的根目录和当前目录设为文件系统根目录。

### 10.3.4 挂装一般文件系统

完成了根文件系统的挂装后，就可以开始挂装其他文件系统。其中的每个文件系统都必须有自己的挂装点(mount point)，这是系统目录树上现有的一个目录。

超级用户可以使用 mount 和 umount 命令来显式地安装和卸载一个文件系统。一个典型的挂装命令是

```
$mount -t ext2 /dev/hda5 /mnt/hda5
```

其中，ext2 指出了要挂装的文件系统类型，/dev/hda5 是文件系统所在的磁盘分区，/mnt/hda5 是文件系统的挂装点。

当对一个文件系统进行 mount 操作时，VFS 需要进行一系列操作来完成文件系统的挂装。

(1) 搜索文件系统类型注册链表 file_systems，从中查找含有该类型名(ext2)的节点。这个节点是 struct file_system_type 结构，其中的成员 get_sb 指向的函数用于读出要安装的文件系统所在的磁盘超级块。

(2) 内核必须准备安装点的 inode 索引节点。它搜索 inode cache 中的 hash 链表，并判断是否可以安装。只有是目录类型才是合格的安装点。

(3) 向 super_blocks 链表申请一个空闲的元素。

(4) 在完成了上述的必要准备工作之后，内核调用 get_sb 所指向的函数，读取要安装的文件系统的磁盘超级块，并把内容写入内存的 super_block 中。

(5) 检查无误后，内核调用 add_vfsmnt()函数，申请一个 struct vfsmount 数据结构，填充其相应数据成员的值后，插入到已挂装文件系统描述符链表 vfsmntlist 的链尾。

VFS 提供 mount()系统调用，其对应的内核响应函数是 sys_mount()，该函数又调用 do_mount()完成实际的挂装工作。

### 10.3.5 卸载文件系统

使用 umount 命令可以卸载已安装的文件系统。卸载前要对该文件系统进行一系列检查。首先检查该文件系统中的文件或目录是否还在使用，并检查其 super_block.s_dirty 成员是否置位，即检查该文件系统是否被修改过，如果已修改，则应先将修改过的内容写回到原物理设备中。当一切检查无误后，内核才卸载该文件系统，将对应的 VFS 的 super_block 超级块释放，并将代表它的 vfsmount 结构从文件系统链表 vfsmntlist 中删除。

VFS 的 umount()系统调用的内核响应函数是 sys_umount()，该函数先调用 umount_dev()执行，后者又通过调用 do_umount()函数完成实际的卸载工作。

## 10.4 进程与文件系统的联系

### 10.4.1 系统打开文件表

在访问文件之前，进程必须首先打开文件，系统调用 open()返回给用户一个文件描述符，它是一个小整数，充当打开文件的句柄。进程通过该文件描述符才能与相对应的文件或物理设备相关联。进程必须用该描述符作为参数才能调用 read 和 write 等系统调用，因为在文件描述符中含有指向一个系统打开文件对象(struct file)的指针和其他一些参数。在系统打开文件对象中含有读写文件的当前位置的偏移量以及指向文件索引节点的指针等。因此由 struct file 结构组成的链表叫做系统打开文件表。

关于 struct file 数据结构和相应的链表，已经在 10.2.2 节中做过介绍，这里就不再重复了。

### 10.4.2 用户打开文件表

文件只有通过进程才能得到执行和被访问操作。而对每个进程来说，需要记录这个进程当前打开的所有文件的信息。在进程描述符中，files 域是进程打开文件表，用来记录和控制进程当前打开的文件。这是一个 file_struct 结构，它的主要成员包括：

```
struct files_struct {
    atomic_t count;                          /* 共享该表的进程数目 */
    spinlock_t file_lock;                    /* 用于表中字段的读/写自旋锁 */
    int max_fds;                             /* 文件对象的当前最大数目 */
    int max_fdset;                           /* 文件描述符的当前最大数目 */
    int next_fd;                             /* 已经分配的最大文件描述符号加 1 */
    struct file * * fd;                      /* 指向文件对象指针数组的指针 */
    …
    struct file * fd_array[NR_OPEN_DEFAULT]; /* 文件对象指针的初始化数组 */
};
```

进程打开文件表中的 fd 字段是一个指向文件对象的指针数组，该数组的长度存放在 max_fds 字段中。

对于在 fd 数组中的每个元素对应一个的文件对象。对这些文件对象来说，数组的索引

就是文件描述符(file descriptor)。通常,数组的第一个元素(索引为 0)是进程的标准输入文件,数组的第二个元素(索引为 1)是进程的标准输出文件,数组的第三个元素(索引为 2)是进程的标准错误文件。

Linux 进程将文件描述符作为主文件标识符。需要注意,两个文件描述符可以指向同一个打开的文件,也就是说,数组的两个元素可能指向同一个文件对象。

当进程开始使用一个文件对象时,它调用内核提供的 fget()函数。这个函数接收文件描述符 fd 作为参数,返回在 current→files→fd[fd]中的地址,即对应文件对象的地址,如果没有任何文件与 fd 对应,则返回 NULL。在第一种情况下,fget()把引用计数器 f_count 的值增 1。

当进程完成对文件对象的使用时,调用内核提供的 fput()函数。该函数将文件对象的地址作为参数,并减少文件对象引用计数器 f_count 的值。另外,如果这个字段变为 0,函数就调用文件操作的 release 方法,释放相关的目录项对象和文件系统描述符,减少索引节点对象的 i_writecount 字段的值(如果该文件是可写的)。最后,将文件对象从"在使用"链表移到"未使用"链表。

### 10.4.3 进程的当前目录和根目录

在 UNIX 和 Linux 系统中,每一个进程都有一个当前工作目录和它自己的根目录。这是进程状态的一部分,以便用户既可以使用相对路径名,也可以使用绝对路径名来访问所需要的文件。

在进程描述符中的 fs 域就是用来维护这两个数据。这是一个指向 struct fs_struct 数据结构的指针,其中包含指向根目录和当前工作目录的目录项结构,以及当前工作目录和根目录的文件系统对象。

## 10.5 ext2 文件系统

### 10.5.1 ext2 文件系统的存储结构

在 Linux 中,ext2 文件系统是最经典的文件系统,它是一个可扩展的、功能较强大的文件系统。内核使用 ext2 文件系统作为它的根文件系统。下面主要以 ext2 文件系统为例,说明 Linux 中文件系统的实现。

在计算机中,程序和数据以文件的形式在存储设备中保存。存储设备包括磁盘、磁带和光盘等。通常情况下使用的存储设备是磁盘。在 Linux 系统中,一个物理存储器可包含一个或多个文件系统。

文件系统由每块 512B 或 512B 的任意倍数所构成的逻辑块序列组成。在同一个文件系统中,这些逻辑块的大小完全相同。块长的选取将直接影响设备与主存之间的数据传输速率和内存的存储能力。大的块长将使得内存和设备之间的数据传输更加容易,但反过来又使得内存页面长度增加,从而影响内存的有效存储能力。

在 ext2 文件系统中,逻辑块的大小(block-size)可以是 1024B、2048B 或 4096B,通常使用 4096B。

在传统的 UNIX 文件系统中，inode 节点集中存放在文件系统的开始处（也就是磁盘开始的位置），而用后面的磁道存放文件数据块。这样，即使是访问一个很短的文件，也必须两次访问磁盘，一次读索引节点，然后再读文件数据块，在两次磁盘操作中需要花费较长的寻道时间，这就形成了文件系统性能的瓶颈。为解决这个问题，需要对文件系统的存储结构进行改造。在 ext2 文件系统中，磁盘分区再被分成多个块组，每个块组由一个或多个连续的柱面组成。每个柱面组都有描述本组磁盘块状态信息的超级块（super block），同时还有各自的索引节点表和空闲块表。这样文件系统就可以把与索引节点相关的文件数据和该索引节点存放在同一柱面组内，从而减少磁盘寻道时间，提高效率。在文件被创建时，先任选一个节点，并优先在该节点所在的柱面组中分配数据块。如果该柱面组不存在空闲的数据块，就在其相邻的柱面组中分配。

此外，在传统的 UNIX 文件系统中，超级块只存放在分区开始的位置，也就是引导块的后面。为增强文件系统的可靠性，即在磁盘发生错误的时候能够恢复系统，应该在每个柱面组中都有一个超级块的备份。显然，增强文件系统的可靠性是以浪费磁盘空间为代价的。

ext2 文件系统的磁盘逻辑结构如图 10.1 所示。其中第 0# 块是引导块（boot block）。引导块中装有引导或初启操作系统的引导代码。在有多个文件系统的计算机系统中，至少有一个文件系统的引导块中装有引导代码。

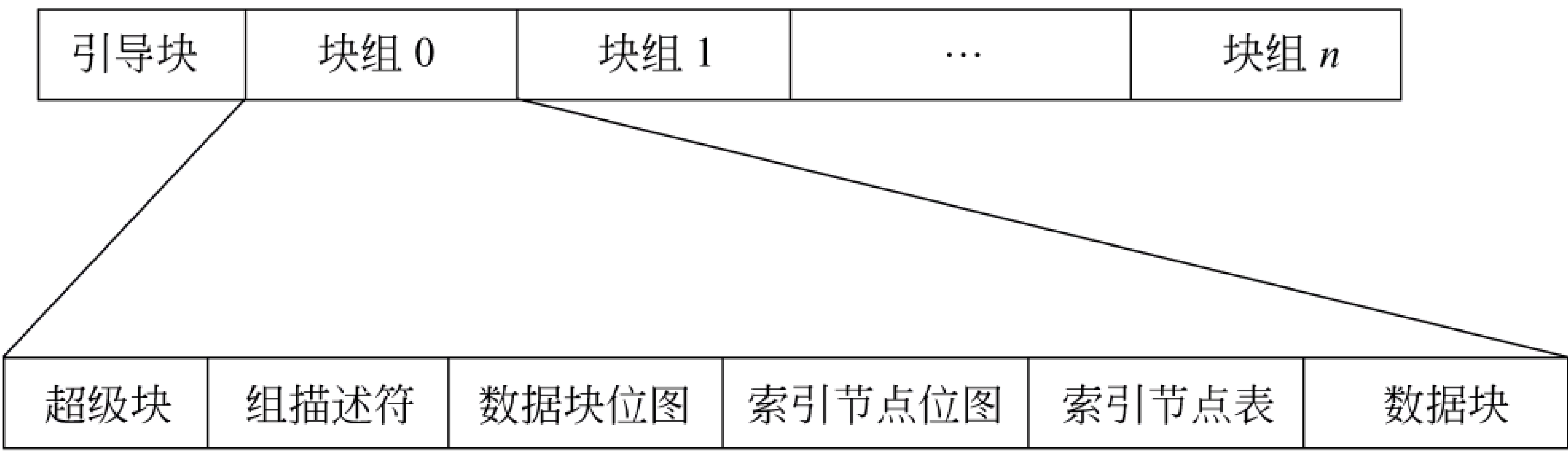



图 10.1　ext2 文件系统逻辑磁盘结构

在图 10.1 中可以看到，ext2 文件系统将一个逻辑分区分为若干个块组（block group）。每个块组中都包含该文件系统的综合信息，即超级块和组描述符。此外，还有该块组的数据块位图、索引节点（inode）位图、索引节点表以及数据区。

在加载文件系统时，实际上只有块组 0 的超级块和组描述符被内核引用，而其他块中的超级块和组描述符只作为备份。

## 10.5.2　ext2 文件系统主要的磁盘数据结构

### 1. ext2 文件系统的磁盘超级块 ext2_super_block

在 ext2 文件系统的每个块组中都保存一个超级块。在块设备作为文件卷安装时，内核将块组 0 的超级块的内容读到内存超级块中，以使得对文件系统的操作能在内存进行。描述磁盘上 ext2 超级块的数据结构是 struct ext2_super_block。下面说明这个结构的主要成员。

字段 s_inode 存放索引节点的个数，而 s_blocks_count 存放 ext2 文件系统的总块数。字段 s_log_block_size 用 2 的幂次表示块的大小，以 1024 为单位。因此，0 表示 1024B 的块，1 表示 2048B 的块，等等。

s_blocks_per_group、s_frags_per_group 与 s_inodes_per_group 字段分别存放每个块组中的块数、片数及索引节点数。

```
struct ext2_super_block {
    __le32  s_inodes_count;             /* 索引节点总数 */
    __le32  s_blocks_count;             /* 文件系统总块数 */
    __le32  s_r_blocks_count;           /* 保留的块数 */
    __le32  s_free_blocks_count;        /* 空闲块计数器 */
    __le32  s_free_inodes_count;        /* 空闲索引节点计数器 */
    __le32  s_first_data_block;         /* 第一个数据块号(总是1) */
    __le32  s_log_block_size;           /* 以2的幂次表示的块大小 */
    __le32  s_log_frag_size;            /* 以2的幂次表示的片大小 */
    __le32  s_blocks_per_group;         /* 一个块组中的块数 */
    __le32  s_frags_per_group;          /* 一个块组中的片数 */
    __le32  s_inodes_per_group;         /* 一个块组中的索引节点数 */
    __le32  s_mtime;                    /* 最后一次挂装操作的时间 */
    __le32  s_wtime;                    /* 最后一次写操作的时间 */
    __le16  s_mnt_count;                /* 挂装操作计数器 */
    __le16  s_max_mnt_count;            /* 最大挂装次数 */
    __le16  s_magic;                    /* 魔数 */
    __le16  s_state;                    /* 文件系统状态 */
    __le16  s_errors;                   /* 错误检查时的动作 */
    __le16  s_minor_rev_level;          /* 次版本号 */
    __le32  s_lastcheck;                /* 最后一次文件系统检查时间 */
    __le32  s_checkinterval;            /* 两次检查之间的间隔 */
    __le32  s_creator_os;               /* 创建文件系统的操作函数 */
    __le32  s_rev_level;                /* 版本号 */
    __le16  s_def_resuid;               /* 保留块的默认UID */
    __le16  s_def_resgid;               /* 保留块的默认GID */
    __le32  s_first_ino;                /* 第一个保留的索引节点号 */
    __le16  s_inode_size;               /* 磁盘索引节点的大小 */
    __le16  s_block_group_nr;           /* 这个超级块的块组号 */
    …
    __u8    s_uuid[16];                 /* 128位文件系统标识符 */
    char    s_volume_name[16];          /* 文件系统卷名 */
    …
    __u32   s_reserved[190];            /* 用NULL填充到块末尾 */
};
```

### 2. ext2 的块组描述符

每个块组有自己的组描述符,用来描述每个块组的使用情况。组描述符的数据结构是 struct ext2_group_desc。

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;            /* 块位图的块号 */
```

```
    __le32  bg_inode_bitmap;                /* 索引节点位图的块号 */
    __le32  bg_inode_table;                 /* 第一个索引节点表块的块号 */
    __le16  bg_free_blocks_count;           /* 组中空闲块的个数 */
    __le16  bg_free_inodes_count;           /* 组中空闲索引节点的个数 */
    __le16  bg_used_dirs_count;             /* 组中目录的个数 */
    __le16  bg_pad;                         /* 按字对齐 */
    __le32  bg_reserved[3];                 /* 用 null 填充 */
};
```

**3. 块位图和索引节点位图**

在每个块组中使用了两个块来分别记录本组内各个数据块的使用情况和索引节点表的使用情况，这两个块分别称为数据块位图和索引节点位图。其中数据块位图中的每一位代表一个数据块，该位为 1，表示所代表的数据块正在使用；为 0，则表示该数据块空闲。索引节点 inode 块位图用来记录本组内索引节点表的使用情况，其中某一位为 1，则相应的节点块忙，否则表示空闲。显然在对 ext2 文件系统进行操作时离不开这两个位图。

每个位图必须存放在一个单独的块中。如果块大小为 4096B，那么一个单独的位图可以描述 32 768 个块的状态。

**4. ext2 文件系统的磁盘索引节点 ext2_inode**

ext2 文件系统中的每个文件对应一个描述它的磁盘数据结构，即磁盘索引节点（inode）。磁盘索引节点中包含文件长度、文件位置、所有者、存取权限、创建时间以及上次访问时间等信息。磁盘索引节点的数据结构是 struct ext2_inode。

```
struct ext2_inode {
    __le16  i_mode;                 /* 文件类型及访问权限 */
    __le16  i_uid;                  /* 文件属主的 UID */
    __le32  i_size;                 /* 以字节为单位的文件长度 */
    __le32  i_atime;                /* 最后一次访问时间 */
    __le32  i_ctime;                /* 文件创建时间 */
    __le32  i_mtime;                /* 最后一次修改时间 */
    __le32  i_dtime;                /* 删除时间 */
    __le16  i_gid;                  /* 文件属主的 GID */
    __le16  i_links_count;          /* 链接数 */
    __le32  i_blocks;               /* 文件的数据块数 */
    __le32  i_flags;                /* 文件标志 */
    union {
        struct {
            __le32  l_i_reserved1;
        } linux1;
        struct {
            __le32  h_i_translator;
        } hurd1;
        struct {
            __le32  m_i_reserved1;
        } masix1;
```

```
    } osd1;                          /* 特定的操作系统的信息 */
    __le32  i_block[EXT2_N_BLOCKS]; /* 数据块指针 */
    …
};
```

其中，文件模式 i_mode 表示文件类型(普通文件、目录文件、块设备、字符设备或 FIFO 文件)和访问权限，而用户标识符 i_uid 以及组标识符 i_gid 定义对该文件具有存取权的用户集合，与该索引节点链接的文件数 i_links_count 表示有多少个不同的文件名指向该文件。另外，该文件所用的物理块指针是一个数组 i_block[EXT2_N_BLOCKS]，它指明文件数据安放在逻辑盘上的位置。有关文件数据的盘块安放与寻址的方法将在后面介绍。

索引节点表由一串连续的块组成，每个 ext2 磁盘索引节点占用 128B。因此，一个长 4096B 的块可存放 32 个磁盘索引项。每个块组中的索引节点总数存放在磁盘超级块的成员 s_inodes_per_group 中。

### 10.5.3 ext2 文件系统的内存数据结构

ext2 文件系统的磁盘组织和内核组织并不完全相同。这主要是因为使用磁盘和内存的考虑角度不同。对于磁盘上存储的数据结构来说，主要是考虑如何节省磁盘空间；而对于内存中存储的数据结构来说，主要是考虑系统运行的效率和性能。因此，ext2 的内存索引节点和磁盘索引节点的数据结构不完全相同。ext2 磁盘数据结构 ext2_super_block 和 ext2_inode 分别对应 ext2 内存数据结构 ext2_sb_info 和 ext2_inode_info。

在内核挂装 ext2 文件系统时，ext2 分区上的磁盘数据结构中的大部分信息被读入，填写到内存中相应的数据结构中。然后，内核通过对位于内存中的超级块和索引节点的操作来实现文件系统的各种功能。

**1. ext2 的内存超级块**

当内核安装 ext2 文件系统时，首先从磁盘上读取磁盘超级块 ext2_super_block 结构体的内容，填充类型为 ext2_sb_info 的内存超级块，并将后者的指针写入 VFS 超级块的 s_fs_info 域，完成从虚拟文件系统到 ext2 文件系统的转变。磁盘超级块内容一旦被读入内存，将一直保留，直到该文件系统被卸载。

为了保证内核能够找出挂装的 ext2 文件系统相关内容，内存超级块 ext2_sb_info 包含下列信息：

- 磁盘超级块字段的大部分内容；
- 块位图高速缓存；
- 索引节点位图高速缓存；
- s_sbh 指针，指向磁盘超级块所在缓冲区的首部；
- s_cs 指针，指向磁盘超级块所在的缓冲区；
- 组描述符的个数；
- s_group_desc 指针，指向组描述符所在的缓冲区的首部的数组；
- 其他与安装状态、安装选项等相关的数据。

**2. ext2 的内存索引节点**

同样地，当 VFS 要处理属于 ext2 文件的索引节点时，从磁盘上读取 ext2_inode 结构，

填充到索引节点描述符 ext2_inode_info 数据结构中。该结构包含下列信息：

- 整个 VFS 索引节点；
- 在磁盘索引节点结构中而不在一般的 VFS 索引节点对象中的大部分字段；
- 片的大小和片数(还未使用)；
- 索引节点所在块组的 i_block_group 块组索引；
- i_prealloc_block 和 i_prealloc_count 字段，在为数据块进行预分配中使用；
- i_osync 字段，是一个标志，表示是否同步地更新磁盘索引节点。

### 10.5.4 数据块寻址

每个非空的普通文件都由一组数据块组成。由于 ext2 文件的数据块在磁盘上不一定是相邻的，因此，ext2 文件系统必须提供一种方法，用这种方法可以在磁盘上建立每个文件块号与相应逻辑块号之间的关系。

磁盘索引节点 ext2_inode 的 i_block 字段是一个有 EXT2_N_BLOCKS 个元素的数组。EXT2_N_BLOCKS 的默认值为 15。这个数组实现了文件块到磁盘逻辑块的转换，如图 10.2 所示。这个数组表示一个大型数据结构的初始化部分。数组的 15 个元素有 4 种不同的类型：

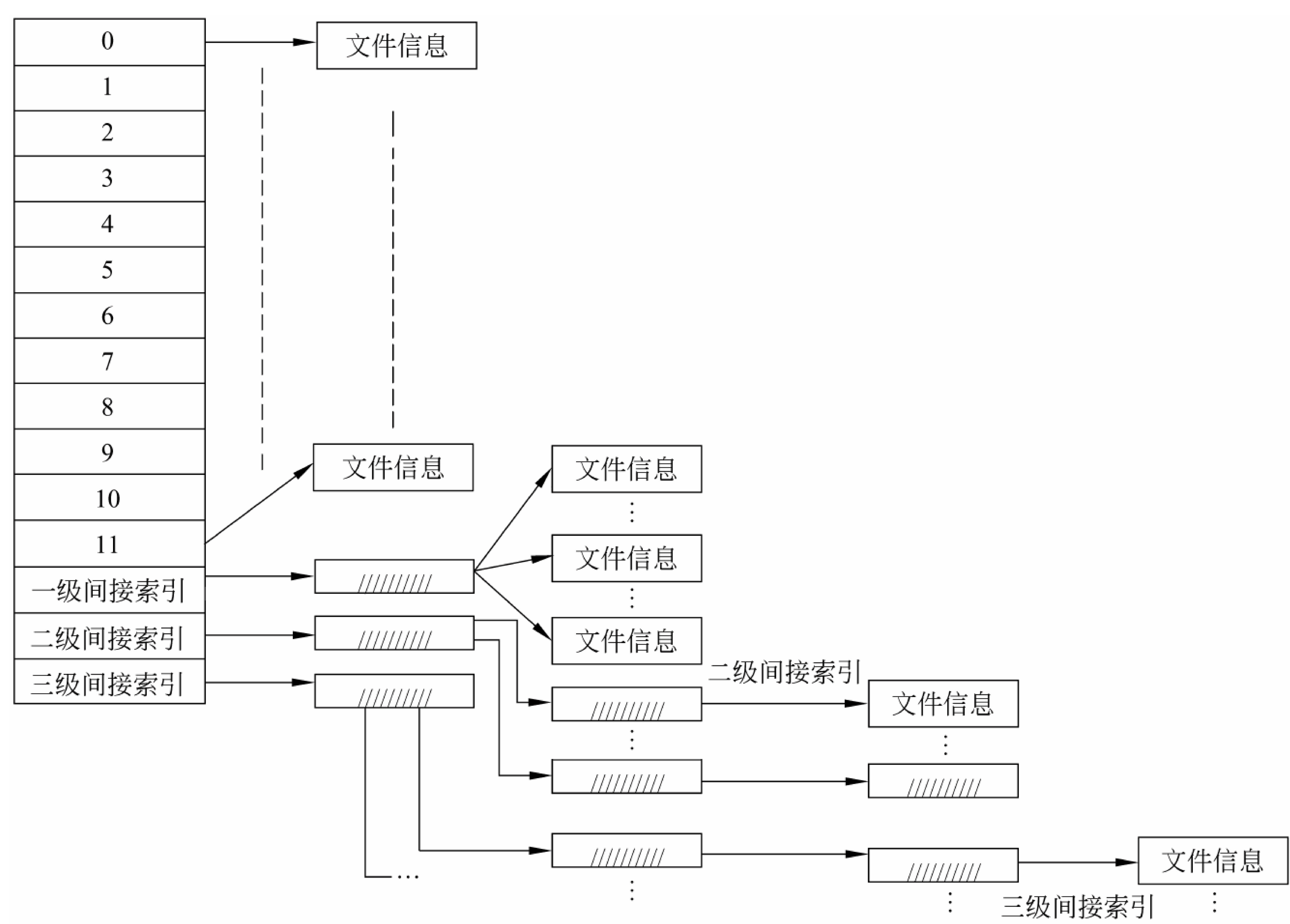



图 10.2　文件映射关系

- 最初的 12 个元素包含的逻辑块号与文件最初的 12 个块对应，即对应的文件块号为 0～11。
- 下标 12 的元素包含一个块的逻辑块号，这个块表示逻辑块号的一个二级数组。这个数组的元素对应的文件块号从 12 到 $b/4+11$，这里 $b$ 是文件系统的块大小(每个

逻辑块号占 4B,因此需要除以 4)。因此,内核为了查找指向数据块的指针必须先访问这个元素,然后,用另一个指向最终块(包含文件内容)的指针访问那个块。

- 下标 13 的元素包含一个块的逻辑块号,而这个块包含逻辑块号的一个二级数组,这个二级数组的数组项依次指向三级数组,这个三级数组存放的才是文件块对应的逻辑块号,范围从 $b/4+12$ 到 $(b/4)^2+(b/4)+11$。
- 下标 14 的元素使用三级间接索引,第四级数组中存放的才是文件块号对应的逻辑块号,范围从 $(b/4)^2+(b/4)+12$ 到 $(b/4)^3+(b/4)^2+(b/4)+11$。

如果文件需要的数据块小于 12,那么两次访问磁盘就可以检索到所需的数据:一次是读磁盘索引节点 i_block 数组的一个元素,另一次是读所需要的数据块。对于大文件来说,可能需要三四次的磁盘访问才能找到需要的块。实际上,这是一种最坏的估计,因为索引节点、缓冲区及页高速缓存都有助于极大地减少实际访问磁盘的次数。

文件系统的块大小也会影响寻址机制,因为大的块允许 ext2 把更多的逻辑块号存放在一个单独的块中。表 10.1 显示了每种块大小和每种寻址方式所存放文件大小的上限。在任何情况下,ext2 文件系统都把文件大小的上限置为 2TB－4096B。

**表 10.1　块大小与其寻址方式所存放文件大小的上限对应表**

| 块大小 | 直接索引 | 一级间接索引 | 二级间接索引 | 三级间接索引 |
|---|---|---|---|---|
| 1024B | 12KB | 268KB | 64.26MB | 16.06GB |
| 2048B | 24KB | 1.02MB | 513.02MB | 256.5GB |
| 4096B | 48KB | 4.04MB | 4GB | 2TB |

# 10.6　块设备驱动

Linux 系统中的设备可分为两类,即块设备和字符设备。管理这些设备的程序模块被称为 I/O 子系统。I/O 子系统控制完成进程与外设之间的通信任务。其中 I/O 子系统的核心部分是控制外设的设备驱动程序。本节主要介绍 Linux 系统的块设备驱动原理。

虽然设备文件也在系统的目录树中,但是它们和普通文件以及目录文件有根本的不同。当进程访问普通文件时,它会通过文件系统访问磁盘分区中的一些数据块;而在进程访问设备文件时,它只要驱动硬件设备就可以了。VFS 的责任是为应用程序隐藏设备文件与普通文件之间的差异。

为了做到这一点,VFS 在设备文件打开时改变其默认文件操作。因此,VFS 把设备文件的每个系统调用都转换成与设备相关的函数的调用,而不是对文件系统相应函数的调用。

## 10.6.1　设备配置

在 Linux 系统中,每一类设备都有自己的驱动程序。而且,每个设备都有自己唯一的设备名,并能像文件那样对其进行存取操作。因此,每个设备都作为特殊文件在文件系统目录树中占据一个节点,只是其索引节点的类型与普通文件不同而已。不过,当文件系统中增加一个普通文件时,可以用系统调用 create 来创建该文件。那么,怎么样来创建一个设备特殊文件呢?

进入 Linux 目录树的/dev 目录，可以看到在里面已经准备好了常用的设备文件。目前，越来越多的 Linux 发行版开始使用 udev 技术，当内核增加一个新的设备驱动模块时，会自动在/dev 目录中创建相应的设备文件。

如果一个新设备和系统连接，需要建立新的设备文件，就要依靠系统管理员使用相应的命令，例如 mknod 来建立特殊文件。

mknod 命令要求管理员提供文件名、文件类型（块设备或字符设备）、主设备号和次设备号。例如：

```
mknod /dev/tty1 c 4 1
```

创建一个设备名为/dev/tty1 的虚拟终端设备（c 代表字符设备文件，b 代表块设备文件），该设备的主设备号是 4，次设备号是 1。在 Linux 系统中，主设备号指示一种设备类型，而次设备号则表示该类设备的一个单元。

### 10.6.2 设备驱动程序的接口

块设备驱动程序把一个逻辑设备号和块号组成的文件系统地址转换成物理设备上特定的物理块号，并启动物理设备和控制器进行 I/O 传输工作。驱动程序有两个接口：与文件系统的接口以及与硬件的接口。

我们知道，在设备文件上发出的每个系统调用都由内核转化为相应设备驱动程序的对应函数的调用。为了完成这个操作，设备驱动程序必须注册自己。如果设备驱动程序被静态地编译进内核，则它的注册在内核初始化阶段进行；而如果驱动程序作为一个内核模块来编译，则它的注册在模块装入时进行。在后一种情况下，设备驱动程序也可以在模块卸载时注销自己。已经注册的设备驱动程序存放在一个散列表中。

用于块设备文件的默认的文件操作方法如表 10.2 所示。

**表 10.2　用于块设备文件的默认操作方法**

| 方法 | 用于块设备文件的函数 | 方法 | 用于块设备文件的函数 |
|---|---|---|---|
| open | blkdev_open() | write | generic_file_write() |
| release | blkdev_close() | mmap | generic_file_mmap() |
| llseek | block_llseek() | fsync | block_fsync() |
| read | generic_file_read() | ioctl | blkdev_ioctl() |

另外，硬件与驱动程序的接口是由机器的有关控制寄存器或操纵设备的 I/O 指令以及中断向量组成的。对硬件 I/O 的监控可以采用轮询模式或中断模式。当一个设备控制器发出中断请求时，系统识别发出中断请求的设备，并调用适当的中断处理程序。

驱动程序与文件系统和硬件的接口如图 10.3 所示。

对于块设备，当文件系统调用有关系统调用时，系统从文件描述符 fd 找到对应的 VFS 内存索引节点。从索引节点信息中可以检查文件类型，从索引节点中抽出主设备号和次设备号。使用主设备号和次设备号，可以从散列表中查找出块设备驱动程序。

接着，系统调用设备驱动程序所提供的 open 方法，打开对应的设备。该驱动程序使调用进程和被打开设备之间建立联系，并初始化其他驱动程序用的数据结构。返回之前当然还要检查打开的合法性，例如，不能同时有几个进程打开同一个打印机设备进行读写操作等。

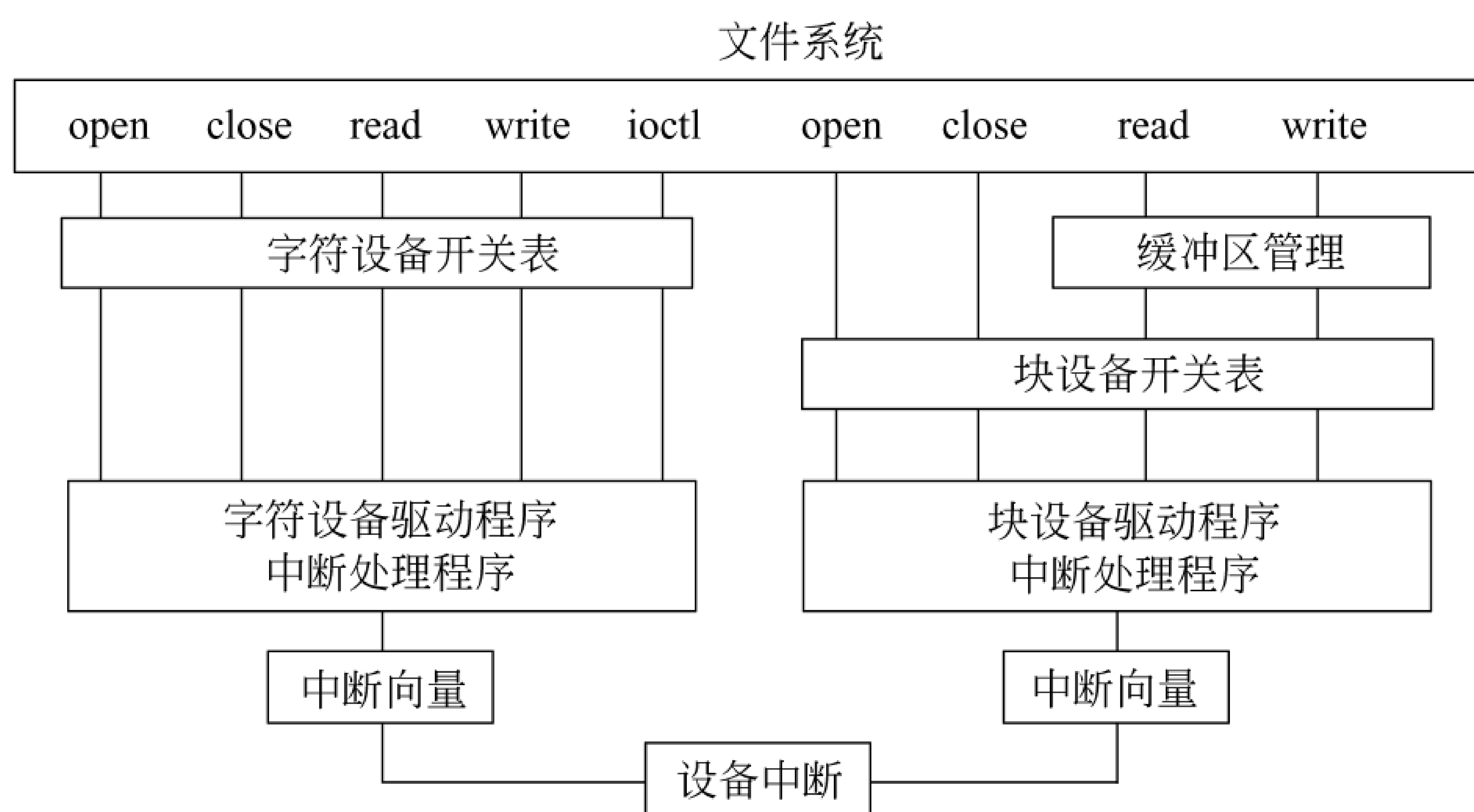



图 10.3　驱动程序及其接口

打开一个设备相当于为一个进程分配一个设备。显然，在使用缓冲池的文件系统中，除了刚开始时需要打开磁盘设备之外，在缓冲区和磁盘块之间进行 I/O 传输时不必打开设备。

系统调用 close 断开用户进程与一个设备的连接。注意，只有当没有其他进程打开此设备时才可以调用关闭过程，否则会引起混乱。

关闭一个设备文件相当于释放一个进程所占有的设备。

块设备有两种基本的 I/O 数据传送方式，分别是块 I/O 操作和页 I/O 操作。函数 bread()从块设备读取一个单独的块，并存放在缓冲区中。它首先检查缓冲区中是否已经有所需要的数据，如果没有，调用 ll_rw_block()函数开始读操作。同时，调用 wait_on_buffer()函数进行等待，这个函数把 current 进程插入 b_wait 等待队列，并挂起，直到缓冲区开锁。

设备与驱动程序的通信方法依赖于硬件。一般计算机中大都设置有状态控制寄存器和数据缓冲寄存器。在数据缓冲寄存器的数据发送完毕之后，状态控制器将会通过总线发出传输完成中断信号，从而引起系统执行一个中断处理程序。至于执行哪一个中断处理程序，则要根据发出中断的设备和中断向量来决定的。一般来说，中断向量所对应的中断处理程序都是针对一类设备的，系统在调用设备中断处理程序时必须将设备号和其他有关参数传递给它，以便识别引起中断的特定设备单元。

总之，块设备驱动程序必须把由逻辑设备号和块号组成的文件逻辑地址翻译成块设备上的特定物理地址，并启动块设备和相应的控制器来进行 I/O 传输工作。在传输完成时还要接收中断信号并完成相应的中断处理。

## 10.7　字符设备驱动

字符设备是指在 I/O 传输过程中以字符为单位进行传输的设备，例如键盘、打印机等。在 UNIX/Linux 系统中，字符设备以字符特殊文件的方式在文件目录中占据一个席位并拥有相应的索引节点。与块设备一样，索引节点中的文件类型指明该索引节点所说明的是一个字符文件，用户可以用与块设备相同的方式打开、关闭和读写字符特殊文件。

字符设备是 Linux 设备中最简单的一种。应用程序可以和存取文件相同的系统调用来

打开、读写及关闭它。与块设备相似，字符设备的驱动程序也需要在内核中注册。内核使用字符设备的主设备号和次设备号来索引设备驱动程序的散列表。

用于字符设备的文件操作只有一个，即打开文件操作 chrdev_open。这个函数使用字符设备的设备号在散列表中查找对应元素，重写文件对象的 f_op 字段。然后，这个函数调用相应的驱动程序的 open 方法来初始化驱动程序。

一旦字符设备文件被打开，则通常用于读和/或写访问；为了做到这点，文件对象的 read 和 write 方法执行设备驱动程序的适当函数。大多数设备驱动程序也通过 ioctl 方法支持 ioctl()调用，该调用允许把特殊的命令发往基本的硬件设备。

## 本章小结

Linux 系统中所有的文件被组织到一个统一的树形目录结构，除了树形目录结构外，Linux 文件系统还具有 4 个特点，包括文件是无结构的字符流式文件，文件可动态地增长或减少，文件数据可由文件拥有者设置相应的访问权限而受保护，外部设备也被看成文件。Linux 文件可以分为普通文件、目录文件、设备文件（字符设备文件和块设备文件）、有名管道（FIFO）、软链接和 UNIX 域套接字 6 种类型。

为了支持多种不同的文件系统，Linux 内核使用虚拟文件系统（VFS）框架，由超级块（superblock）、索引节点（inode）、文件（file）和目录项（dentry）等对象组成。VFS 是位于程序和具体文件系统之间的软件层，它隐藏了各种硬件的具体细节，实现了与文件系统相关的所有系统调用，为各种文件系统提供了通用接口，应用程序只需与 VFS 进行交互，而不用关心文件系统的实现细节。借助于 VFS，不同的文件系统可通过注册、挂载加入到 Linux 系统中，被用户所使用。目前，Linux 可为多种类型的文件系统提供支持，包括基于磁盘的文件系统、基于网络的文件系统和特殊的文件系统等。

在访问文件之前，进程必须首先打开文件，通过该文件描述符与相对应的文件或物理设备相关联，files 域用来记录和控制进程当前打开的文件。当进程开始使用一个文件对象时，调用内核提供的 fget()函数获得对应文件对象的地址，调用 fput()函数或 release 方法释放相关目录项对象和文件系统描述符，将文件对象从“在使用”链表移到“未使用”链表。每个进程都有一个当前工作目录和它自己的根目录，并可以采用相对路径名或绝对路径名来访问所需要的文件。

ext2 文件系统是 Linux 系统中最经典的文件系统，它是一个可扩展的、功能较强的文件系统。在 ext2 文件系统中，磁盘分为块组，每个块组由一个或多个连续的柱面组成，每个柱面组由一个超级块（superblock）描述本组磁盘块状态信息，同时还有各自的索引节点表和空闲块。描述超级块的数据结构是 ext2_super_block；每个块组中有描述块组使用情况的数据结构 ext2_group_desc，数据块位图和索引节点位图分别记录本块组各数据块和索引节点表的使用情况；数据结构 ext2_inode 描述了每个文件的磁盘数据结构，即磁盘索引节点（inode）。另外，ext2 文件系统中还设置了内存超级块和内存索引节点来描述文件在内核中的存储情况。

块设备和字符设备都是以特殊文件的形式在 Linux 文件系统中存在，因此，在使用块设备或字符设备时，首先要将驱动程序在内核中注册，然后利用系统提供的虚拟文件系统接口

来访问。

## 习　　题

10.1　Linux 文件系统的特点是什么？简单描述一个典型的 Linux 系统的目录结构。

10.2　Linux 的文件类型有几种？分别是什么？

10.3　什么是 VFS？它有什么作用？其通用数据模型是什么？

10.4　VFS 包括哪些系统调用？分别简述其功能。

10.5　简述文件系统的注册、挂装以及卸载过程。

10.6　ext2 文件系统的数据块寻址是如何实现的？

10.7　Linux 系统中的设备可分为几种？分别是什么？

# 第 11 章　Windows 的设备管理和文件系统

Windows 设备管理的对象包括除处理器和内存之外的大多数硬件设备，如交互设备、外存设备和终端设备等。计算机的存储器由内存和外存组成，处理器可以通过总线控制器直接对内存进行寻址访问，但需要通过设备驱动才能对外存进行操作。因此，Windows 将外存看作一种特殊的设备，外存管理是 Windows 设备管理的重要组成部分。在外存管理的基础之上，Windows 构建了多种格式的文件系统。

## 11.1　Windows I/O 系统的结构

Windows 管理的设备主要是完成计算机和外界进行交互的输入/输出设备，因此将操作系统的设备管理服务称为 I/O（输入/输出）系统。本节介绍 Windows I/O 系统的设计目标及设备管理服务的系统结构。

### 11.1.1　设计目标

Windows I/O 系统为应用程序和操作系统服务提供了一个操作设备的抽象层，它由若干个运行在核心态的系统服务组成。可以从 Windows I/O 系统的设计目标来了解它的主要特点：

- 为所有的设备提供统一的安全和命名机制，以方便这些资源的共享。
- 提供基于包交换的高效异步 I/O 请求处理机制，以提高应用与设备交互的效率。
- 为使用高级语言编写设备驱动程序提供支持，以便在不同体系结构的计算机之间移植驱动程序。
- 支持可扩展的分层驱动程序结构，以方便对设备管理功能的扩充和修改。
- 支持驱动程序的动态加载和卸载，以最大程度地节省系统资源。
- 提供即插即用的功能，使得系统能够自动侦测新的硬件设备，并自动分配必要的系统资源。
- 提供电源管理功能，使得系统和相关的设备在不使用时可以进入低功耗状态。
- 支持多个可安装的文件系统，包括文件分配表文件系统（FAT）、CD-ROM 文件系统（CDFS）、通用磁盘格式文件系统（UDF）和 NT 文件系统（NTFS）。

### 11.1.2　设备管理服务

如图 11.1 所示，由多个运行在核心态的系统服务与设备驱动程序一起完成设备管理，它们在体系结构上可以分为 3 个层次：最低层是硬件抽象层，它为操作系统服务提供了一个对主板上的设备的逻辑封装，如处理器和中断控制器等；在其上为设备驱动层，它为计算机操作外接硬件设备提供了一个逻辑接口；再往上是设备管理层，它实现对所有计算机外部设备的管理，并为用户态的应用提供操作设备的接口。

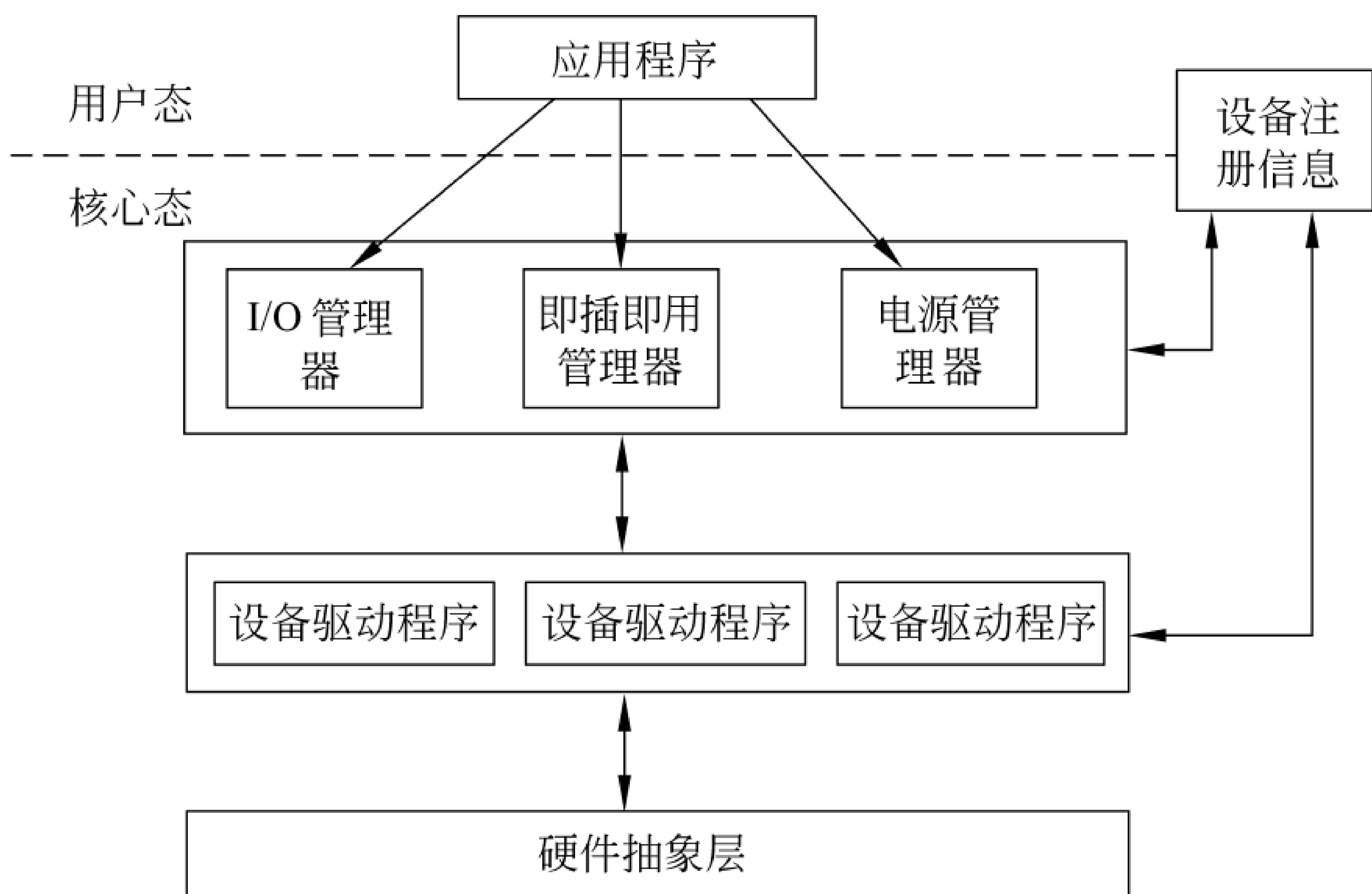



图 11.1　Windows 设备管理体系结构

下面简单地介绍构成设备管理服务的各部分的功能。

**1. I/O 管理器(I/O Manager)**

它是 Windows I/O 系统的核心,提供支持 Windows 设备驱动程序的系统结构。它将应用的 I/O 请求传递到相应的设备驱动程序,并将处理的结果返回给应用。

**2. 即插即用管理器(PnP Manager)**

它和 I/O 管理器以及总线管理驱动一起,检测硬件设备的加入和移出,并分配相应的硬件资源。

**3. 电源管理器(Power Manager)**

它和 I/O 管理器以及相应的设备驱动程序一起,管理该设备的能耗状态。

**4. 设备驱动程序(Device Driver)**

它为访问特定的设备提供一个 I/O 接口,设备驱动从 I/O 管理器得到处理指令,在处理完后会通知 I/O 管理器。如果需要其他的设备驱动协助完成指令,设备驱动程序会通过 I/O 管理器将指令转到相关的设备驱动程序。

**5. 注册表(Registry)和 INF 文件**

注册表记录了和系统相连的硬件设备描述以及初始化和配置信息,INF 文件是用来安装相关设备驱动程序的文件。

**6. 硬件抽象层(Hardware Abstraction Layer,HAL)**

它为设备驱动程序提供一个和计算机主板硬件无关的抽象层,使得设备驱动程序可以通过统一的接口来访问计算机主板上的设备,而不用考虑不同计算机在硬件配置上的差别。

## 11.2　设备驱动程序和 I/O 处理

设备驱动程序是 Windows I/O 系统中直接和硬件设备交互的部分,设备管理层的服务通过它们来完成具体的 I/O 请求处理过程。本节介绍设备驱动程序的分类和基本构成,并通过对 Windows I/O 处理的过程分析来说明它的工作原理。

### 11.2.1 设备驱动类型和结构

如上节所述，大部分的设备驱动程序运行在核心态，可以按照它们不同的性质分为下面的 3 个大类：

（1）文件系统驱动。Windows 的文件系统是通过驱动程序来实现的，它构建在外存管理的基础之上。它接受对文件操作的请求，并将这些请求翻译成具体的外存设备的操作命令。

（2）即插即用设备驱动。它们和硬件设备直接交互，并和 Windows 即插即用管理器一起工作，来完成对设备的管理。这类驱动管理的常见设备有大容量存储设备、视频适配器、输入设备和网络适配器等。

（3）非即插即用设备驱动。这类驱动不直接和硬件设备打交道，而是为用户态的应用访问核心态的服务和驱动提供接口。这类驱动又被称作内核扩展，常见的这类设备驱动有网络接口和协议驱动。

Windows 设备驱动是一组例程的集合，I/O 控制器通过调用它们来完成 I/O 请求的处理。在操作系统中，设备驱动由一个驱动对象来表示，I/O 管理器在调入设备驱动时创建该对象。一个典型的 Windows 设备驱动常常包括下面的例程：

- 初始化例程。当 I/O 管理器将设备驱动程序调入操作系统时，会调用该例程。该例程通过填写相应的系统数据结构来注册该设备驱动的其他例程，并作必要的全局初始化。
- 设备加入例程。所有的即插即用设备都需要实现一个设备加入例程，当即插即用管理器检测到一个新的硬件加入时，会调用该例程。该例程为新的设备分配一个新的设备对象。
- 调度例程。设备驱动程序通过调度例程来实现主要的功能。例如，设备、文件系统和网络设备实现的打开、关闭、读取和写入等功能例程。
- I/O 启始例程。当一个 I/O 请求处理开始时，I/O 管理器通过该例程来初始化与设备交换的数据。
- 中断服务例程。当一个设备发出中断请求时，系统的中断调度器将控制转到该例程。它用来实现需要实时处理的设备请求，并将剩下的操作留给中断服务延迟过程调用例程处理。
- 中断服务延迟过程调用例程。在中断服务例程返回调用程序后，中断服务延迟过程调用例程有更多的时间处理完 I/O 请求的操作，以减少对其他的系统服务的影响。
- 终止例程。在分层设备驱动中需要实现这一例程，当低层的设备驱动完成处理后，会调用该例程来通知上层设备驱动相应的处理结果。
- 调出例程。Windows 的设备驱动程序可以动态地调入和调出，当调出设备驱动程序时，I/O 管理器通过调用该例程来释放该驱动程序占用的系统资源，并将驱动程序移出内存。

### 11.2.2 Windows 的 I/O 处理

Windows 应用程序通过调用相应的子系统函数来发出 I/O 请求，I/O 请求通过 I/O 管

理器和设备驱动程序到达实际的物理设备。下面我们将在介绍不同类别的 I/O 请求的基础之上,分析 Windows 处理 I/O 请求的基本过程。

**1. I/O 处理的类型**

常见的 Windows I/O 处理可以按照其功能分为以下几种类型:

(1) 同步 I/O 和异步 I/O:当驱动程序初始化 I/O 处理后会马上返回 I/O 管理器,同步 I/O 操作会等待设备处理完数据,再返回到调用程序继续执行;异步 I/O 操作则会马上返回到调用程序,等 I/O 请求处理完后,再进行数据同步。

(2) 快速 I/O:为了提高系统访问文件或高速缓存的速度,Windows 还提供了一种直接访问文件系统驱动和缓存管理器的 I/O 处理机制。这种 I/O 处理避免了发送 I/O 请求包而带来的延时,可以提高访问的效率。

(3) 映射文件 I/O:通过映射文件,Windows 可以将磁盘上的文件当作进程的虚拟空间的一部分。应用可以将文件当作一个大的数组来直接访问,而内存管理器通过映射文件 I/O 来完成映射文件到磁盘文件的转换。在核心操作系统服务中,映射文件 I/O 被用于高速文件缓存和映像激活(调入可执行程序)。

(4) 集中式的 I/O:为了提高文件访问效率,Windows 的 I/O 系统提供了一种集中访问连续文件块的访问机制。当应用程序对连续的文件块进行读写时,即使它们在进程的地址空间中不是连续存放的,I/O 系统也会将几个 I/O 请求合成一个来进行处理,以便减少磁盘 I/O 的次数。

**2. 单层驱动的 I/O 处理**

Windows I/O 系统通过 I/O 请求包来表示一个 I/O 请求,在应用调用一个 I/O 服务时,I/O 管理器会创建一个 I/O 请求包,并将它保存在非分页系统内存空间中,以便核心态系统服务对它进行访问。

通常一个 I/O 请求是由一层或多层设备驱动程序配合完成的。如图 11.2 所示,首先分析一个简单的单层驱动处理同步 I/O 请求的过程:

(1) 应用调用 I/O 处理函数,并通过相应的子系统动态连接库发出的 I/O 请求;

(2) 子系统的动态连接库调用 I/O 管理器的请求服务;

(3) I/O 管理器分配一个 I/O 请求包来表示该请求,并将它发送到相应的设备驱动程序;

(4) 设备驱动程序将 I/O 请求包中的数据发送到相关的设备,进行 I/O 操作;

(5) 在完成 I/O 处理后,设备发出中断通知处理器操作完成;

(6) 处理器的控制转到驱动程序的结束服务来通知 I/O 管理器处理完毕,并由 I/O 管理器将控制返回到调用程序。

**3. 多层驱动的 I/O 处理示例**

如图 11.3 所示,以文件系统的磁盘操作 I/O 为例,分析多层驱动处理一个非同步 I/O 请求的过程。

当 I/O 管理器收到一个磁盘操作 I/O 请求后,为它分配一个 I/O 请求包并首先将它发送给文件系统驱动程序。文件系统驱动程序再将该请求包转发给磁盘驱动。

与单层驱动不同的是,多层驱动的 I/O 请求包有一组驱动栈信息,每一层驱动通过维护相应的驱动栈信息来完成自己的操作并进行信息传递。

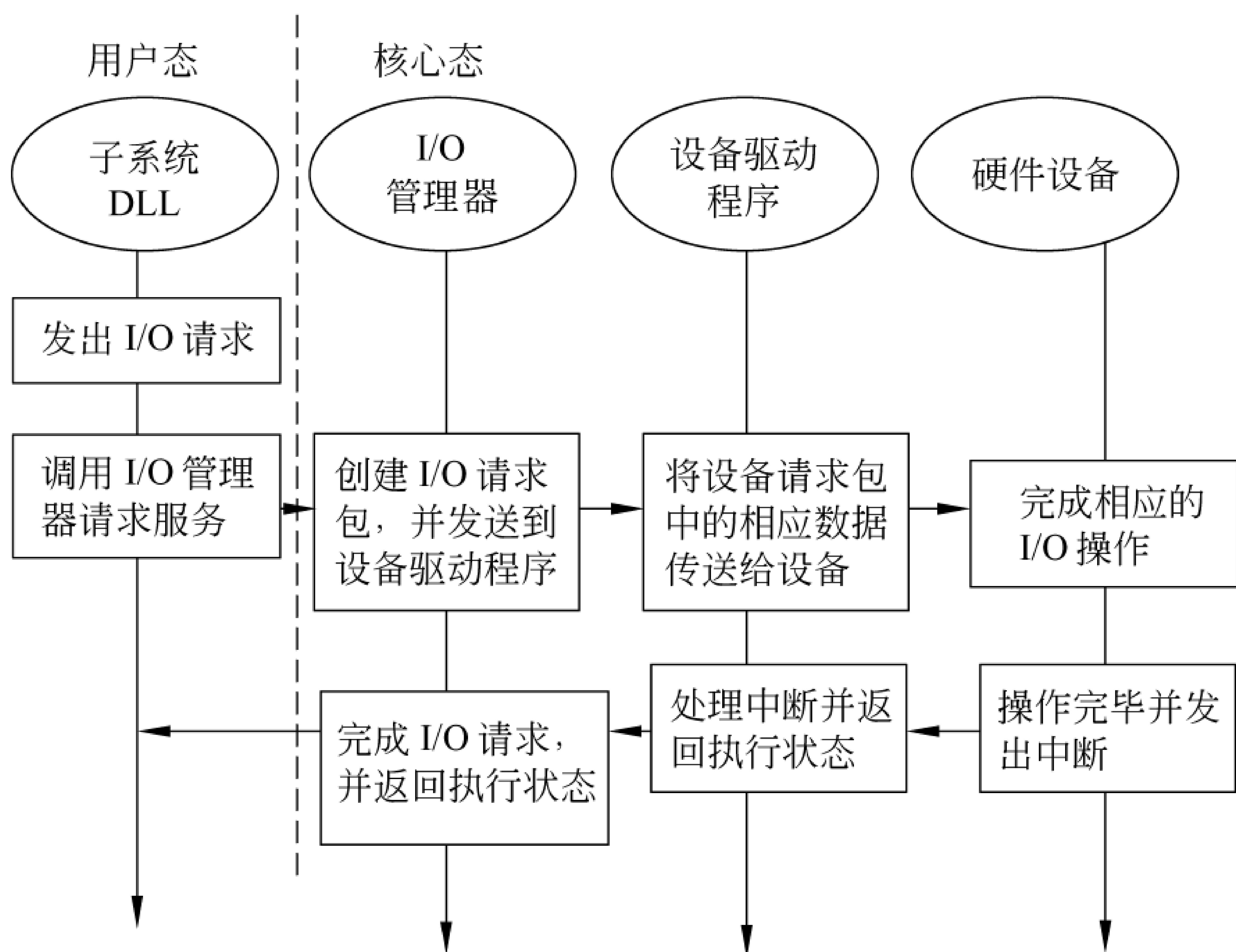



图 11.2　单层驱动处理同步 I/O 请求的过程

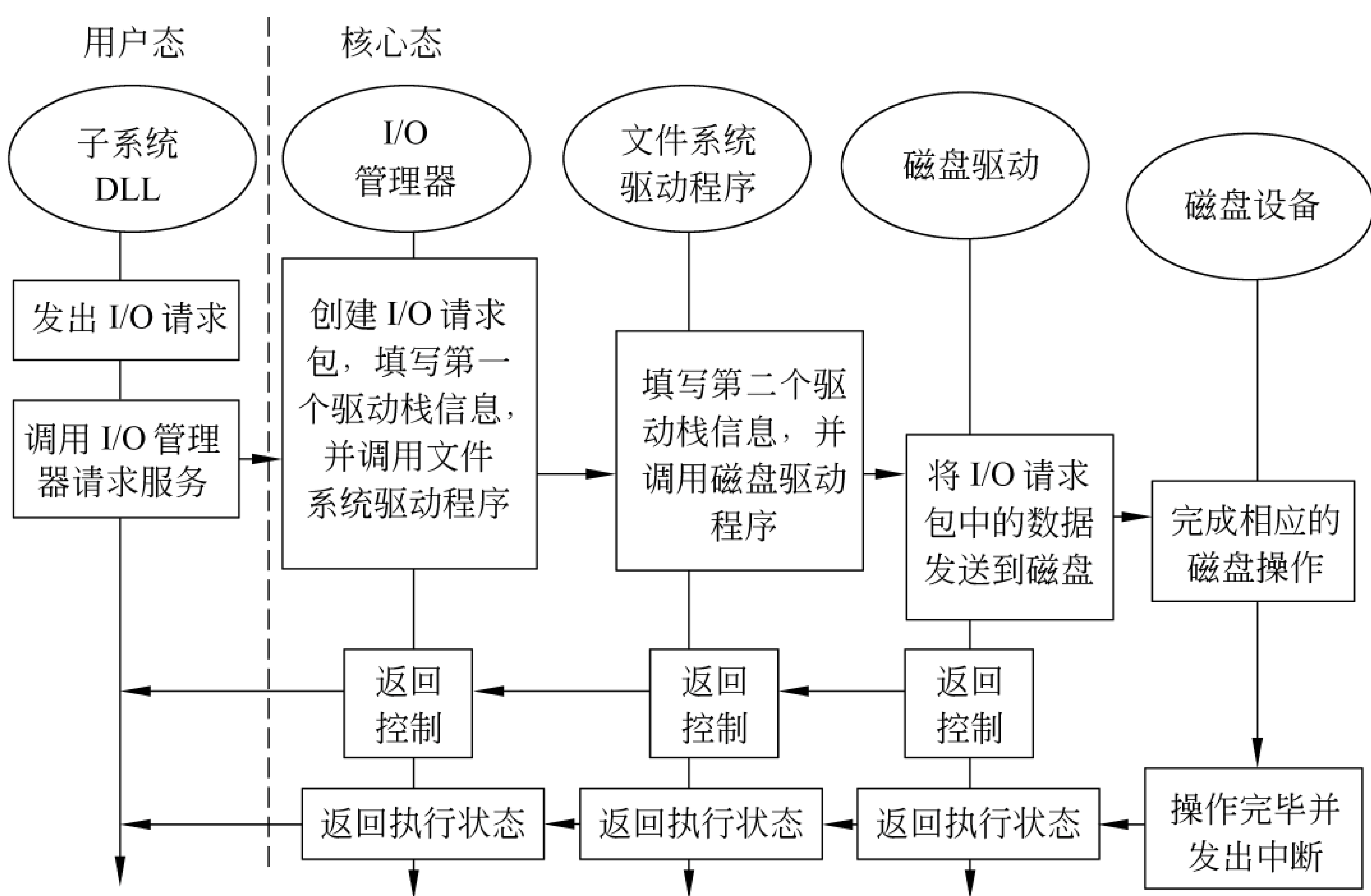



图 11.3　多层驱动处理非同步 I/O 请求的过程

和同步 I/O 处理不同，异步 I/O 处理将 I/O 请求包传递到磁盘驱动后，磁盘驱动在将相关数据发送到磁盘设备的同时立即将控制返回到调用程序。等到磁盘设备完成操作后，再通过中断将执行状态返回到应用程序，并作相关的数据同步。

## 11.3　Windows 的文件系统

在 11.2 节中提到，Windows 的文件系统是以设备驱动的形式来实现的。操作系统将信息以文件的形式存储在外存（如磁盘）上，文件系统驱动和磁盘驱动共同构建了 Windows

的文件系统。本节介绍 Windows 如何在磁盘管理的基础之上,通过文件系统驱动来构建文件系统。

### 11.3.1 Windows 磁盘管理

Windows 将磁盘分为固定大小的“扇区”,扇区的大小取决于不同的存储设备。大部分硬盘的扇区大小为 512B,而 CD-ROM 的扇区大小一般为 2048B。

相邻的扇区集合组成“分区”,Windows 通过分区表来存储每个分区的开始扇区、大小和其他相关的特性。Windows 通过“卷”来抽象一个或几个分区,它是文件系统操作磁盘的逻辑的单元。

如图 11.4 所示,文件系统是构建在磁盘管理之上的。系统通过分区管理器和卷管理器与磁盘设备进行交互,为文件系统提供一个以卷为单位的逻辑视图。文件系统在卷管理器的基础之上构建文件系统的格式,并为 I/O 管理器提供操作文件的接口。

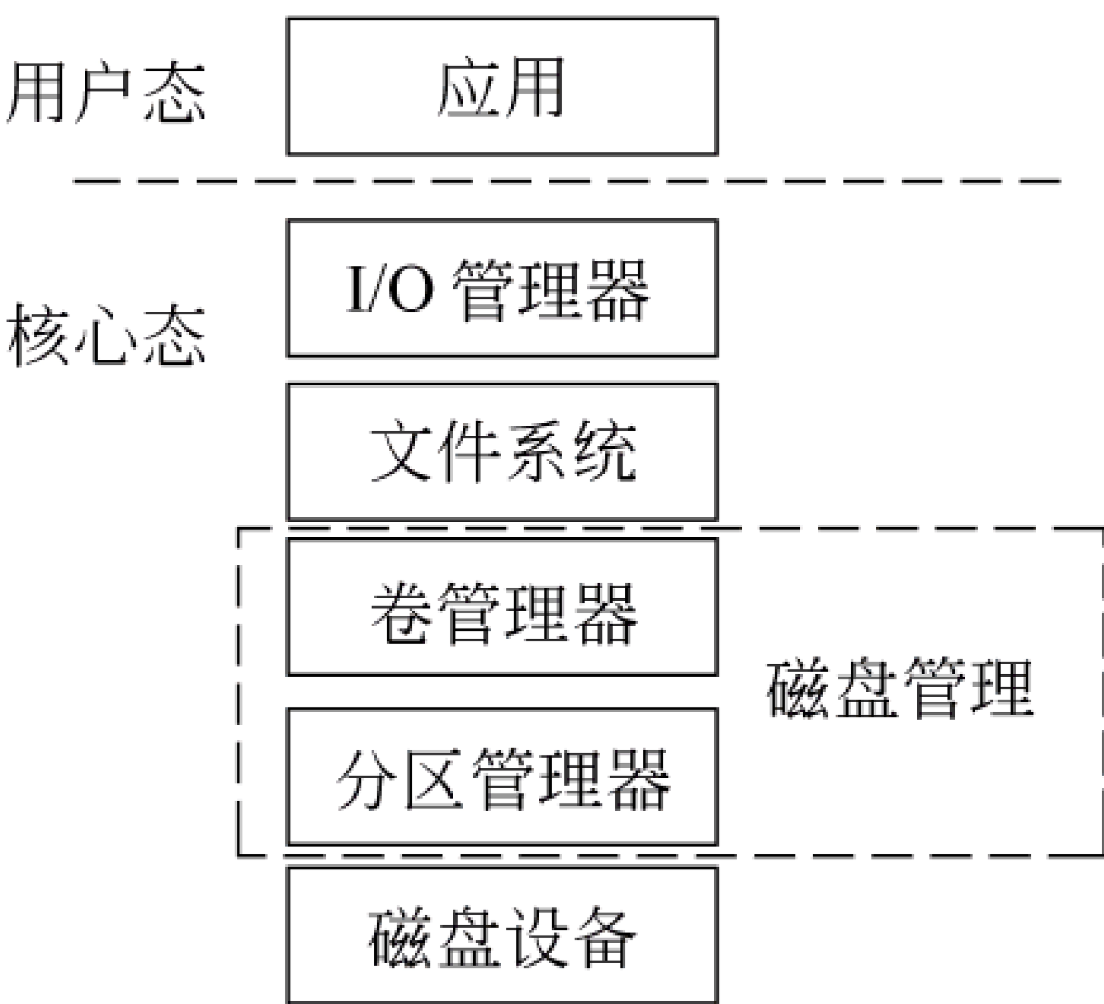



图 11.4　Windows 磁盘管理相关的系统服务

### 11.3.2 Windows 文件系统格式

Windows 支持多种文件系统格式,包括 CDFS、UDF、FAT12、FAT16、FAT32 和 NTFS。

**1. CDFS 和 UDF**

CDFS(CD-ROM 文件系统)是一个支持 CD-ROM 文件的只读文件系统,最大支持的文件大小为 4GB,最多支持 65 535 个目录。UDF(通用磁盘格式文件系统)主要提供了对 DVD 文件的支持。

**2. FAT**

FAT(文件分配表文件系统)是一个简单的文件系统,它最初是为 DOS 操作系统设计的。它适用于小容量的磁盘,文件目录也比较简单。为了向后兼容,Windows NT 体系结构的操作系统仍然支持 FAT 文件系统。

FAT 文件系统是根据其组织形式(文件分配表)而命名的,文件分配表位于卷的开头。为了防止文件系统遭到破坏,FAT 文件系统保存了两个文件分配表,当其中一个遭到破坏时,另外一个可以作为备份。而且,文件分配表和根目录必须放在磁盘的一个固定的位置,这样系统在启动时才可以找到需要的文件。

以 FAT 文件系统格式化的卷以簇为单位进行分配,FAT 有 3 个不同的版本:FAT12、FAT16 和 FAT32,FAT 后面的数字代表用来表示簇号的位数,这个数字,表示在一个分区上可以存储的簇的数目越大。

**3. NTFS**

NTFS (NT 文件系统)是基于 Windows NT 体系结构的 Windows 操作系统的主流文件系统,NTFS 的每个卷可以支持 $2^{32}-1$ 个文件,一个文件最大可达到 16TB。

NTFS 具有许多优秀的性能,如文件和目录的安全机制、磁盘配额、文件压缩以及加密

等，而且当系统不正常中止后，文件系统可以自动恢复目录和文件的结构信息。11.4节将详细介绍NTFS的结构。

### 11.3.3 Windows文件系统驱动

Windows通过文件系统驱动来管理不同格式的文件系统，它运行在核心态。作为设备管理的一部分，文件系统驱动程序需要在I/O管理器中注册。为了提高文件操作的效率，文件系统驱动程序和内存管理器交互得非常密切。为了提高吞吐量，它还和高速缓存管理器直接交互。

文件系统驱动程序在I/O管理器中注册后，I/O管理器可以通过它来识别卷。每一个Windows文件系统的第一个扇区都被预留为卷的根扇区，通过根扇区，文件系统驱动程序可以找到相应的文件系统所需要的数据信息。

当文件系统驱动识别了一个卷后，它会创建一个设备对象来表示文件系统。相应的存储设备管理器也创建一个设备对象来表示具体的存储设备。通过这两个设备对象，I/O管理器将传递到卷设备管理器的I/O请求转发到存储管理器的设备对象。

应用可以通过两种方式来访问文件，一种是直接通过文件访问函数进行文件操作，如ReadFile和WriteFile等，它通过磁盘驱动的I/O处理来操作磁盘。另外一种是通过读写映射文件的地址空间来完成文件操作，它通过映射文件来操作磁盘。

## 11.4 NTFS文件系统

作为Windows主流的文件系统，NTFS可以满足企业级应用的需要。例如，在系统掉电或系统崩溃时，NTFS可以保证数据的一致性；NTFS具有统一的安全机制，保证敏感的数据不被非法访问；NTFS提供了低成本的软件数据冗余以保护用户数据等。

### 11.4.1 NTFS的特点

NTFS具有以下特点。

(1) 可恢复性。NTFS利用事务来实现文件系统的可恢复，操作系统将一系列对文件系统结构修改的操作组成事务。事务不间断地执行完后可以保证文件系统的一致性，如果事务在没有执行完之前被中断，已执行的操作必须进行回退，以保证文件系统结构恢复到事务操作之前。

(2) 安全性。NTFS的安全性是通过Windows的对象模型来实现的，每一个打开的文件都由一个文件对象来表示，文件对象的安全描述作为文件的一部分存储在磁盘上。在一个进程打开一个文件对象的句柄时，Windows的安全系统服务会验证该进程是否有访问该对象的授权。

(3) 冗余与容错。数据的可恢复性可以保证文件系统在磁盘上可以访问，但不能保证恢复所有的用户文件。NTFS通过冗余来保护关键的文件系统信息，当存储该信息的磁盘扇区损坏时，系统可以利用冗余扇区来恢复文件数据。

(4) 动态坏簇重映像。当系统试图读一个被损坏的扇区时，Windows容错驱动程序会自动搜索到备份扇区。同时，NTFS会重新分配一个扇区，作为该扇区新的备份。

（5）基于统一字符编码(Unicode)的命名机制。NTFS 用统一字符编码来存储文件名、目录名和卷名，16 位的字符编码可以让不同国家的用户用自己的语言来命名，使得 Windows 的国际化版本很容易实现。

（6）文件压缩。NTFS 支持文件压缩功能，并透明地管理压缩和解压的过程。用户还可以选择将一个文件目录压缩，系统会将该目录下的所有文件压缩。

（7）加密。NTFS 的安全性并不能保护计算机丢失或文件被泄露后的数据私密性，NTFS 通过加密文件系统来实现敏感数据的加密保护。

### 11.4.2 NTFS 的磁盘结构

下面介绍 NTFS 如何划分磁盘空间，以及如何组织文件和目录。

**1. 卷**

系统格式化磁盘时，首先将磁盘分割成一个或多个卷。卷代表着磁盘的一个逻辑分区，每一个卷包含一系列的文件和可被利用的磁盘空间。

**2. 簇**

簇是在一个卷上的固定大小的磁盘空间，簇的大小在系统格式化时就确定了。簇的大小和卷的大小相关，它一般是物理扇区大小的偶数倍。NTFS 基于簇而不是扇区来操作，是为了使文件系统独立于不同大小的物理扇区。NTFS 可以通过设置较大的簇空间来支持大容量的磁盘，或者通过簇大小的调整来支持非标准物理扇区的磁盘。当磁盘分区很大时，还可以通过设置较大的簇空间来减少磁盘碎片和磁盘定位的时间。

为了定位相关的簇在磁盘上的物理位置，NTFS 引入了逻辑簇号(logical cluster number)。逻辑簇号表示从卷的开始到结束的顺序编号，将相应的逻辑簇号乘以簇大小就得到了该逻辑簇号在物理盘上的位置。NTFS 通过虚拟簇号(virtual cluster number)来定位一个文件中数据的位置，虚拟簇号将文件从开始到结尾顺序编号。每个虚拟簇号都映射到一个逻辑簇号，因此连续的虚拟簇号在物理上不一定是连续存放的。

**3. 文件**

NTFS 的一个特点就是存储在卷上的所有数据都保存在文件中，包括用于定位和查找文件的文件系统的结构数据。当磁盘的一部分损坏后，NTFS 可以重定位这些文件，使得磁盘可以继续使用。NTFS 中，包含目录在内的文件名的长度可达 255B，可以包含 Unicode 字符、空格和句点。

主控文件表是 NTFS 卷结构的核心，它由一组文件记录构成。不管簇的大小如何，每个文件记录的大小固定为 1KB，每一个文件在主控文件表上都有一个文件记录与之对应。除了数据文件之外，NTFS 的卷还包含了用来实现文件系统的结构文件，主控文件表本身也以文件的形式存在。为了区别于用户文件，这些系统结构文件用 $ 标志符作为文件名的首字符，如主控文件表的文件名为 $MFT。

每个文件都通过一个文件记录来描述，文件记录用不同的属性来描述文件的文件名、时间信息和其他文件说明信息，文件数据也可以看作是文件记录的一个属性。

当文件的所有属性(包括数据属性)占用的空间小于 1KB 的文件记录时，所有的数据可以直接存储在文件记录中。如图 11.5 所示，当文件的大小超过 1KB 时，NTFS 会分配额外的盘区来存储文件的数据，并通过文件记录的数据属性指向了额外分配的盘区。

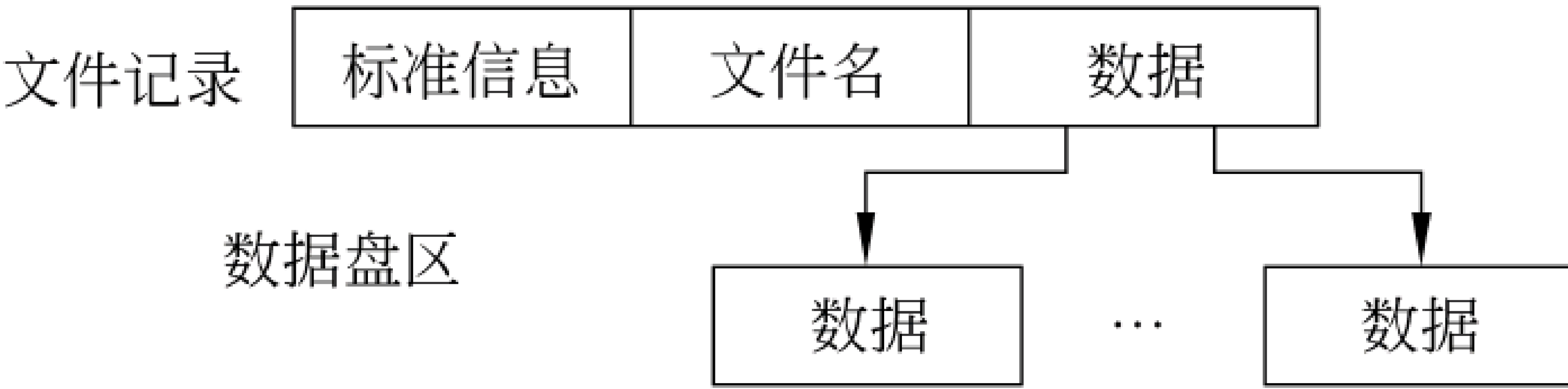



图 11.5　大于 1KB 的文件存储结构

### 4. 目录

在 NTFS 文件系统中，文件目录的结构为该目录所有文件名的索引记录，它也存放在主控文件表中。在创建一个文件目录时，NTFS 会根据该目录下的文件的名称属性建立索引。以一个卷的根目录为例，其文件名索引记录如图 11.6 所示。

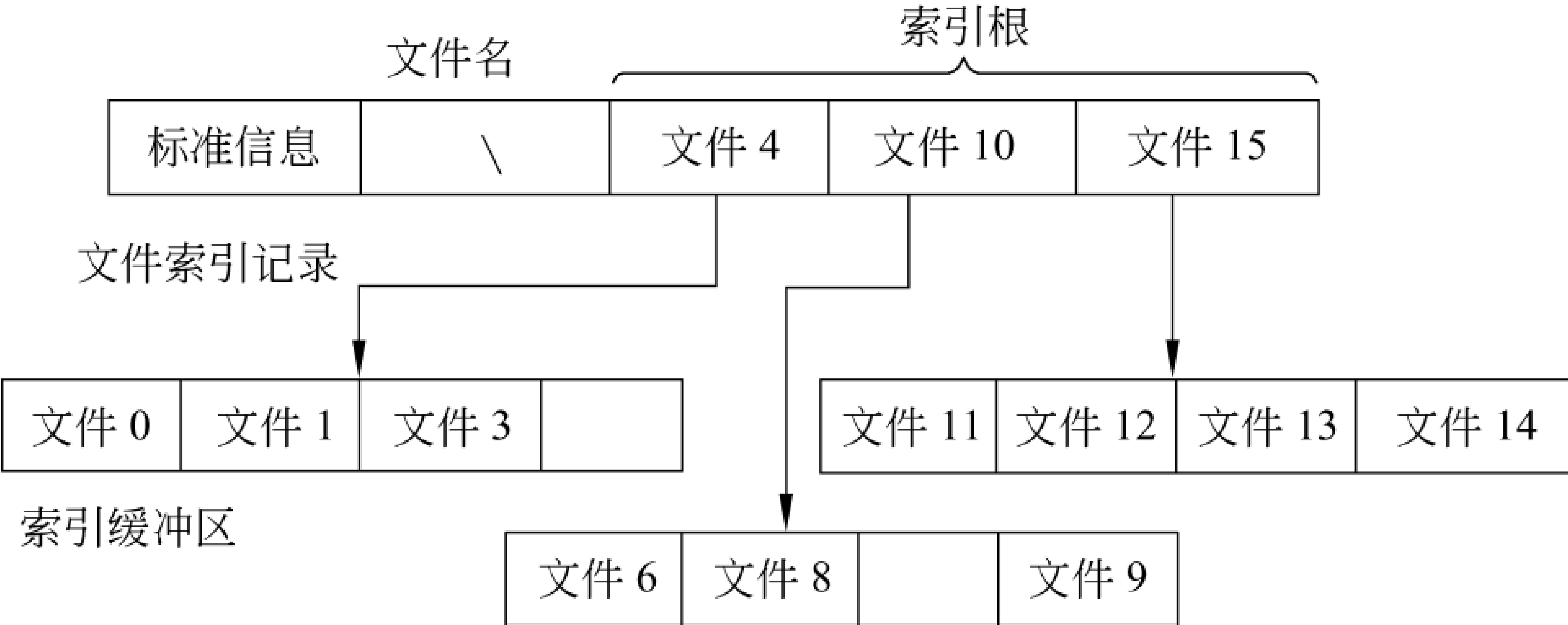



图 11.6　Windows 的根目录存储结构示例

一个目录的主控文件记录将其目录中的文件名和子目录名进行排序，并保存在索引根属性中。对于一个大的目录，系统用一个固定为 4KB 大小的索引缓冲区来存储不能放在索引记录中的文件名。NTFS 用 B+树来实现索引管理，用 B+树来存储文件名有很多好处，由于文件名是排序后存放的，所以目录查找起来非常快。而且，B+树趋向于横向增长，因此当文件目录不断加深时，NTFS 的查找效率不会降低。

## 11.4.3　NTFS 的文件系统恢复

NTFS 利用恢复机制保证当系统掉电和出现错误时，所有的文件系统操作事务都可以执行完毕，从而保证了磁盘卷的完整性。

在前面讲到，NTFS 是使用事务处理来实现恢复机制的。恢复操作仅限于文件系统的数据，以在确保卷的完整性的同时，使得恢复过程很快。对用户文件来说，应用程序需要定义相应的恢复机制来确保文件的完整性。

日志技术在恢复机制中起到关键的作用，NTFS 在执行任何改变文件系统数据结构的操作之前都要将该操作记入一个日志文件中。当系统崩溃后，重启后的系统可以根据日志文件继续完成一个事务中还没有执行的操作，或者取消一个事务中已经完成的操作。

日志中的日志记录通常有两种：一种是更新记录，说明了某一操作在更新了文件系统结构数据后，继续执行或取消执行该操作所在的事务的方法。另一种是断点记录，NTFS 会定时在日志文件中写入这种记录，说明当系统在断点处崩溃时需要执行的恢复操作。

# 本 章 小 结

本章介绍了 Windows 操作系统的设备管理机制，以及作为其重要组成部分的磁盘管理和文件系统。

首先，从计算机外设管理输入/输出的角度介绍了 Windows 设备管理的设计目标。然后详细介绍了构成 Windows I/O 系统的各功能模块以及它们之间的关系。I/O 管理器是 Windows I/O 系统的核心，它一方面为应用程序提供操作设备的接口，另一方面为设备驱动程序提供了支撑环境。

设备驱动程序直接和硬件设备进行交互，为 I/O 管理器访问硬件提供了逻辑接口。本章介绍了常见的设备驱动程序的类型和基本结构。文件系统也是设备驱动程序的一种，它通过磁盘驱动管理物理磁盘上的数据。大部分 Windows 的设备驱动程序都支持即插即用功能，使得 I/O 系统能方便地管理各种外接设备。还有一类设备驱动程序不直接和硬件设备打交道，它们为运行在用户态的应用访问系统内核和驱动程序提供接口。

在介绍了设备驱动程序的结构之后，本章还介绍了 Windows 设备管理系统处理一个 I/O 请求的过程。I/O 请求由用户程序调用相应的 I/O 函数开始，通过 I/O 请求包将请求从 I/O 管理器传递到相应的设备驱动程序。I/O 请求的处理可能由一个设备驱动程序完成，也可能是由多个分层的设备驱动程序一起配合完成。一般来讲，设备驱动程序在完成数据向硬件设备的传送之后，就将控制返回到 I/O 管理器。在同步 I/O 处理中，I/O 管理器会等待硬件设备处理完请求后再返回调用程序；而异步 I/O 处理会马上将控制返回到调用程序，等硬件设备处理完请求后再做数据同步。

在介绍了通用的设备管理和 I/O 请求处理后，本章又介绍了 Windows 的磁盘管理和文件系统。磁盘管理是文件系统的基础，Windows 将磁盘分为固定大小的扇区，又将相邻的扇区组合成分区，最后用卷来抽象分区数据，作为文件系统操作磁盘的逻辑单元。Windows 的分区管理和卷管理统称为磁盘管理，它为 Windows 构建文件系统提供了一个基础。Windows 支持多种文件格式，包括支持 CD-ROM 和 DVD 的文件格式，以及从 DOS 操作系统继承的 FAT 文件格式，但主流的文件格式是 NTFS。

本章详细介绍了 NTFS 文件格式的特点、磁盘管理、构成文件和目录的方式，以及如何实现文件系统的自动恢复。NTFS 在设计上可以满足企业级用户的需要，在支持大容量的文件格式的同时，还考虑到文件系统的安全性、可恢复性、数据的冗余保护以及文件的加密。NTFS 通过可变大小的簇来操作磁盘，比用扇区操作更加灵活。

文件和目录是文件系统的基本表现形式，本章介绍了 NTFS 如何构建文件和目录。NTFS 文件系统的所有数据都被保存在文件中，包括文件系统格式的格式数据。Windows 通过主控文件表来表示文件和目录的组织方式，每个文件和目录都是主控文件表中的一个记录，每个文件记录的大小固定为 1KB。文件记录由若干属性组成，文件数据也是其中的一个属性。当文件的所有属性占用的空间小于 1KB 时，可以用一个文件记录表示该文件；当文件大于 1KB 时，NTFS 会分配额外的盘区来储存文件数据。NTFS 的目录是该目录下所有文件的文件名和子目录名的索引，索引记录也被保存在主控文件表中。一个目录下的文件名和子目录名被保存在索引记录的根属性中，如果目录属性超过 1KB，系统会分配索

引缓冲区来存放这些索引。NTFS 用 B+树的结构来管理这些索引，以提高目录操作的效率。

最后，本章还介绍了 NTFS 利用事务机制来实现文件系统自动恢复的方法。

## 习　题

11.1　Windows 的设备管理系统由哪几部分构成？它们之间的关系如何？

11.2　试述 I/O 管理器的功能，它如何将 I/O 请求传递到设备驱动程序？

11.3　描述 Windows 设备驱动程序的构成以及其中各个例程的功能。

11.4　同步 I/O 处理过程和异步 I/O 处理过程的主要区别是什么？

11.5　Windows 的磁盘管理由哪几部分构成？它们如何管理磁盘？

11.6　分析一个安装了 Windows 操作系统的计算机的文件目录结构。

11.7　为什么 NTFS 用簇而不是扇区来操作磁盘？它的优点是什么？

11.8　描述一个 NTFS 文件的主控表文件记录的格式，并画出它的结构图。

11.9　描述一个大目录结构的主控文件表文件索引记录格式，并画出它的结构图。

11.10　NTFS 的日志中记录了哪几种操作？它们的功能是什么？

# 第12章　嵌入式操作系统简介

嵌入式操作系统(Embedded Operating System,EOS)是一种支持嵌入式系统应用的实时操作系统,是嵌入式系统极为重要的组成部分。本章以VxWorks嵌入式实时操作系统为例来介绍嵌入式操作系统的进程管理、调度、内存管理、设备管理和文件系统的基本原理,并介绍如何进行嵌入式操作系统的开发。

## 12.1　嵌入式操作系统的总体架构

嵌入式操作系统又称实时操作系统(Real Time Operation System,RTOS),是运行在嵌入式硬件平台上,对系统软硬件资源进行统一协调控制的操作系统软件,通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面和标准化浏览器Browser等。

### 12.1.1　嵌入式操作系统特点及分类

**1. 嵌入式操作系统的特点**

目前存在很多种嵌入式操作系统,如VxWorks、μC/OS、嵌入式Linux和WinCE等,这些操作系统功能日益完善,在嵌入式系统中能实现很多桌面通用操作系统具备的功能。嵌入式操作系统除了具有通用操作系统的基本特点之外,更有其特殊的部分,如高实时性、可裁剪性、高可靠性、接口统一、网络功能强大、体积小巧、固化代码、操作简单易学等特点,其中重要的包括高实时性、可裁剪性和高可靠性等。

1）高实时性

大多数嵌入式操作系统工作在对实时性要求很高的场合,如监测控制、数据采集和信息处理等。这些过程往往是一个连续的过程,一个过程必须在一个定长的时间内完成,否则若逻辑和时序出现偏差,将会引起严重后果。因此,嵌入式操作系统中实时性是其重要特征之一。

2）可裁剪性

由于嵌入式系统是面向单一设备的单一应用,其环境复杂多变,如硬件环境中除CPU之外,其他硬件并没有标准化,而系统的应用对功能、可靠性、成本、体积和功耗等有一定需求。因此,嵌入式操作系统必须是开放的、可伸缩的体系结构,其中的很多部件必须具有很强的可裁剪性,便于修改,从而能适应不同嵌入式系统的需求。

3）高可靠性

高可靠性是嵌入式系统的基本特征之一,嵌入式系统中,出于安全方面的考虑,要求系统不能崩溃,而且还要有自愈能力。因此,作为嵌入式系统的最基本软件的操作系统,需要尽可能减少安全漏洞和不可靠的隐患,通过系统监控进程监视各进程的运行状况,在遇到异

常情况时采取措施对其进行修复，实施有利于系统稳定可靠的方法将问题解决，从而从嵌入式系统的底层增强可靠性。

4）统一的接口

嵌入式操作系统可提供各种设备的驱动接口。随着各类嵌入式操作系统的开发，考虑到为嵌入式应用软件的设计者提供统一的服务接口，就必须约定嵌入式系统提供的接口，从而为嵌入式应用软件的运行提供无关性平台。

5）网络功能强大

嵌入式操作系统必须不仅能支持 TCP/IP 协议，还能支持其他如 UDP/PPP 协议等，并为嵌入式系统提供统一的 MAC 访问层接口，从而为各种移动计算设备预留接口，提供对更多的嵌入式系统的支持。

**2. 嵌入式操作系统的分类**

当前，常用的嵌入式操作系统可分为商用系统、专用系统以及开放系统三大类。

1）商用嵌入式操作系统

商用嵌入式操作系统功能较强大，辅助工具较齐全，可应用的范围也较广，在许多领域都有应用，例如 Microsoft 的 Windows CE、WindRiver 的 VxWorks、EPSON 的 ROS33、CoreTek 的 DeltaOS、pSOS＋、3Com 的 PalmOS 以及中科院的 Hopen 等。

2）专用嵌入式操作系统

专用嵌入式操作系统一般不对用户公开，它是一些专业的公司针对该公司产品所特制的嵌入式操作系统。专用嵌入式操作系统功能相对较弱，但具有较强的针对性，而且比普通的商用嵌入式操作系统更加安全可靠。

3）开放嵌入式操作系统

开放嵌入式操作系统是近年来迅速发展的一类操作系统。因为应用系统的开发者可免费得到这些系统的源代码，因此开发难度低。但开放嵌入式操作系统的功能简单，技术支持以及系统的稳定性也相对较差，因此对应用系统开发者提出了较高的要求。

### 12.1.2 嵌入式操作系统的总体架构

嵌入式操作系统通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接口、通信协议、图形界面和标准化浏览器 Browser 等。随着嵌入式系统的发展，嵌入式操作系统的体系结构出现了单块结构、分层结构、微内核结构和构件化结构等多种。

**1. 单块结构**

单块结构是最早出现的一种结构，该结构将整个嵌入式实时操作系统看成一个整块，内部分为若干个模块，模块之间直接相互调用，不分层次，形成网状调用模式，如图 12.1 所示。

由于单块结构本身具有强大功能的完整内核，能为嵌入式软件开发提供非常完整的平台，因此，其具有以下特点：

（1）可设置嵌入式芯片的通用接口。

（2）允许设备驱动器、网络服务器、防火墙等复用一些公共的代码或其他开源代码。

（3）根据功能进行系统的开发，避免实现多余功能以及额外的内存占用。

（4）用户运行的应用程序设计简洁、开发简单、易于调试、相对可靠。

（5）大多数嵌入式系统对实时性要求不高，单块结构可以实现快速响应。

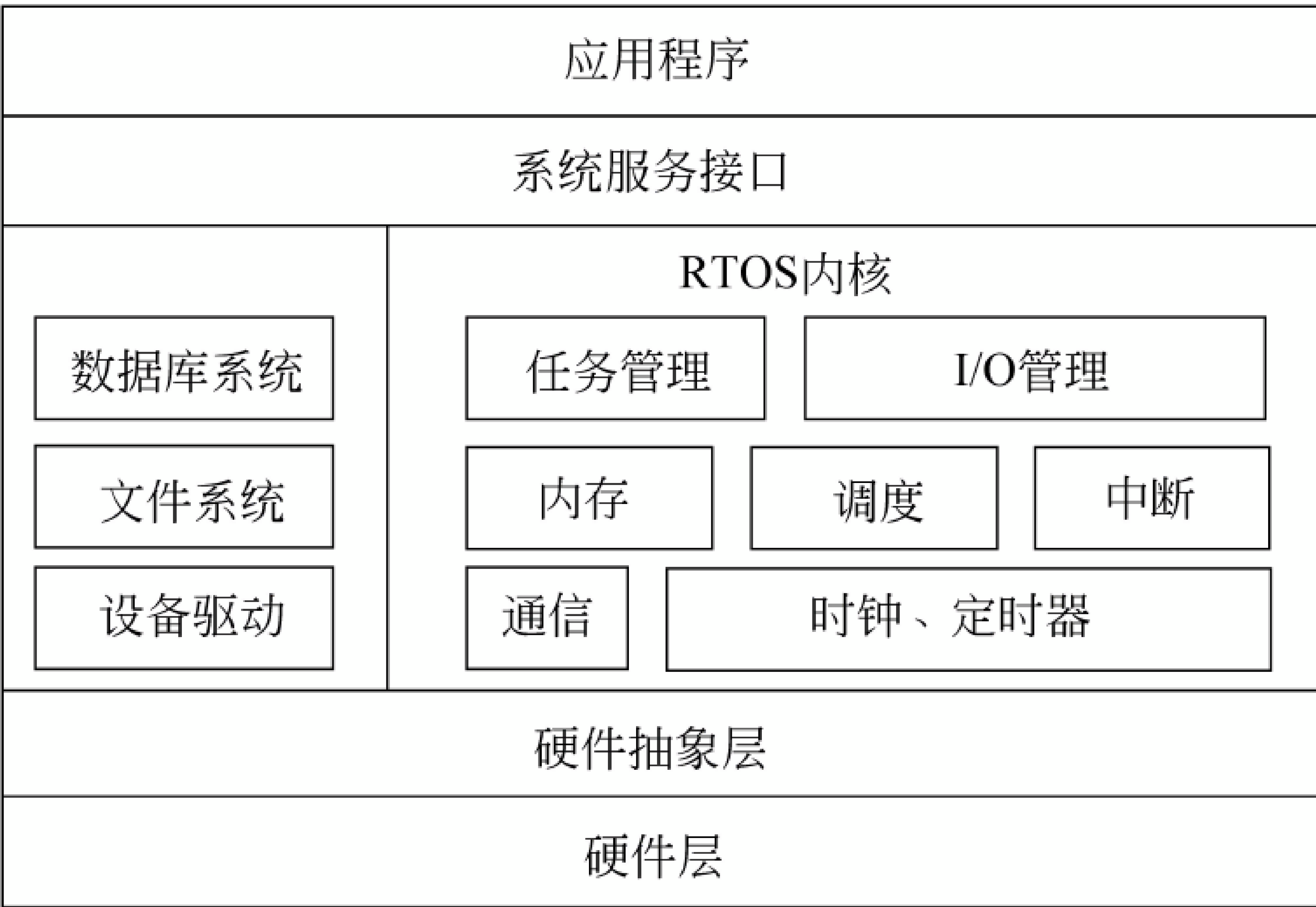



图 12.1　嵌入式操作系统的单块结构

(6) 通过硬件的设计保证对请求的快速响应。

(7) 对简单的小系统,单块结构有几乎最高的系统效率和实时性保障。

(8) 通用 RTOS 系统的单位成本更低。

(9) 对于复杂系统的应用,需大量硬件资源。

(10) 内核的复杂性使系统的运行变得不可预测和不可靠。

(11) 模块越多,模块之间的依赖越严重,在可裁剪性、可扩展性、可移植性和可维护性等方面存在明显缺陷,制约了该类结构 RTOS 的应用。

单块结构常应用在 Windows CE 和嵌入式系统中。

**2. 分层结构**

分层结构是现今许多流行的嵌入式实时操作系统所采用的体系结构,在分层结构中,每一层对其上层好像一个虚拟计算机,下层为上层提供服务,上层使用下层提供的服务;在层与层之间定义良好的接口,上下层通过结构进行交互和通信;每一层中划分为一个或多个模块(组件),并可针对应用需求配置个性化 RTOS。整个分层结构如图 12.2 所示。

分层结构的实时多任务内核部分是整个分层结构的核心,其基本工作是任务切换,其运行与队列密不可分;分层结构中的其他部分是对整个嵌入式操作系统的有力支撑,包括存储管理、I/O 设备管理(包括逻辑 I/O 和设备驱动)、嵌入式文件系统、网络系统和命令解释器等。

分层结构常用于 VxWorks、DeltaOS 等嵌入式操作系统中。

**3. 微内核结构**

微内核结构是目前的主流结构之一,又称为客户/服务器(C/S)结构,在这种体系结构中,微内核仅提供任务调度、任务间通信、底层网络通信、中断处理接口和实时时钟等几种基本服务,且内核非常小,任务在独立的地址空间运行,速度极快;而传统的操作系统提供的其他服务,如存储管理、文件管理、中断处理和网络通信协议等,在内核以上以协作任务的形式出现,每个协作任务可以看成一个功能服务器。微内核结构如图 12.3 所示。

由于微内核结构中将内核功能和其他服务功能分开,因此,微内核结构具有以下优点和不足:

(1) 微内核结构中,其他服务模块可任意裁剪,符合嵌入式操作系统的发展要求。




图 12.2　嵌入式操作系统的分层结构

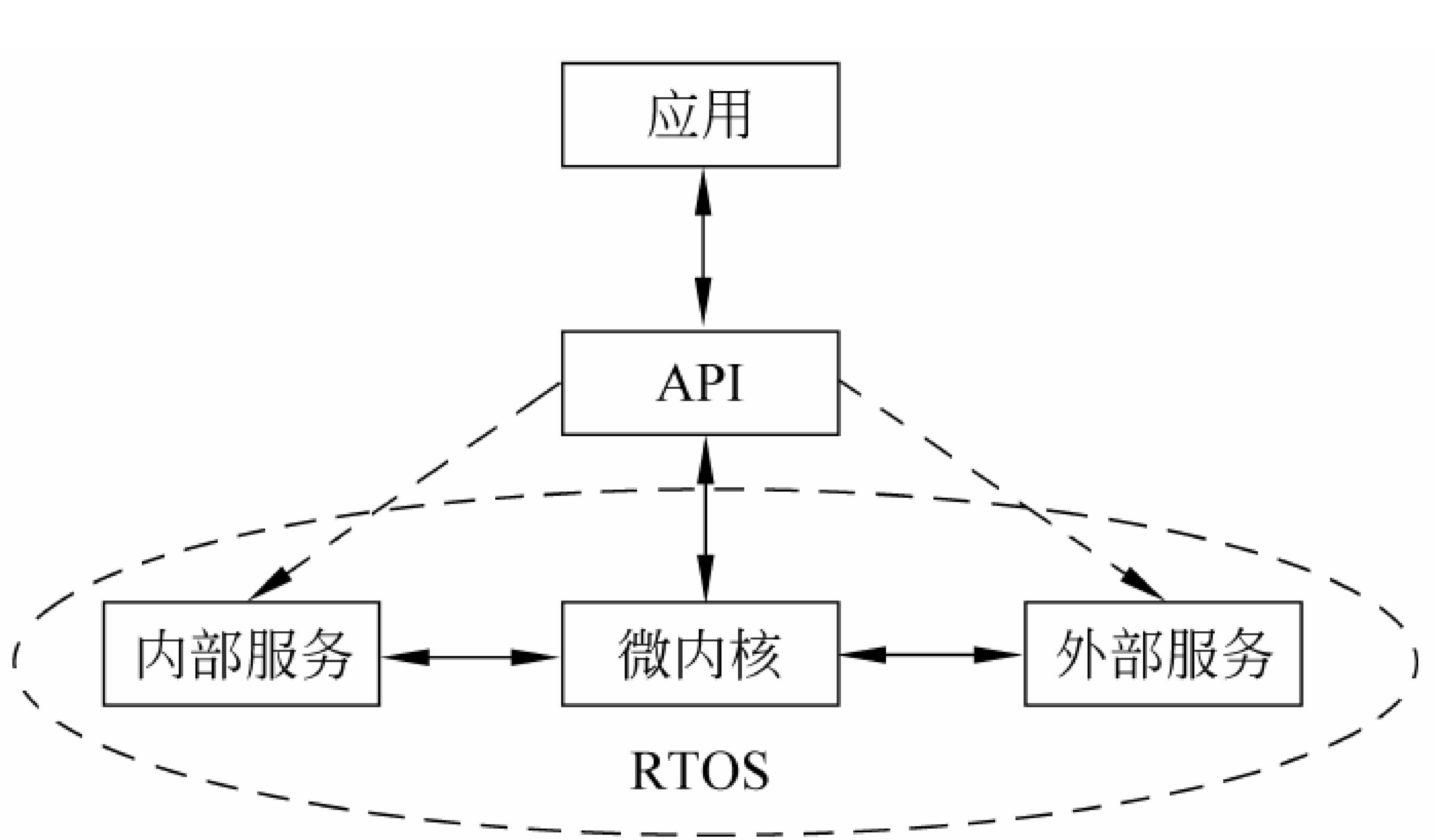



图 12.3　嵌入式操作系统的微内核结构

（2）可以更方便地扩展功能，包括动态扩展。

（3）可以更容易地做到上层应用与下层系统的分离，便于系统移植。

（4）服务模块的可重用性高。

（5）在任务执行时，需要客户端与服务器端通信，会增加一定的开销，与整体系统相比性能将有一定的下降。

微内核结构常用于 QNX 等嵌入式操作系统中。

**4．构件化结构**

构件化结构是采用构件组装思想及技术而设计的体系结构，在构件化结构中，嵌入式操作系统的内核由一组独立的构件和一个构件管理器组成，构件管理器用于维护内核构件之间的协作关系。构件化结构如图 12.4 所示。

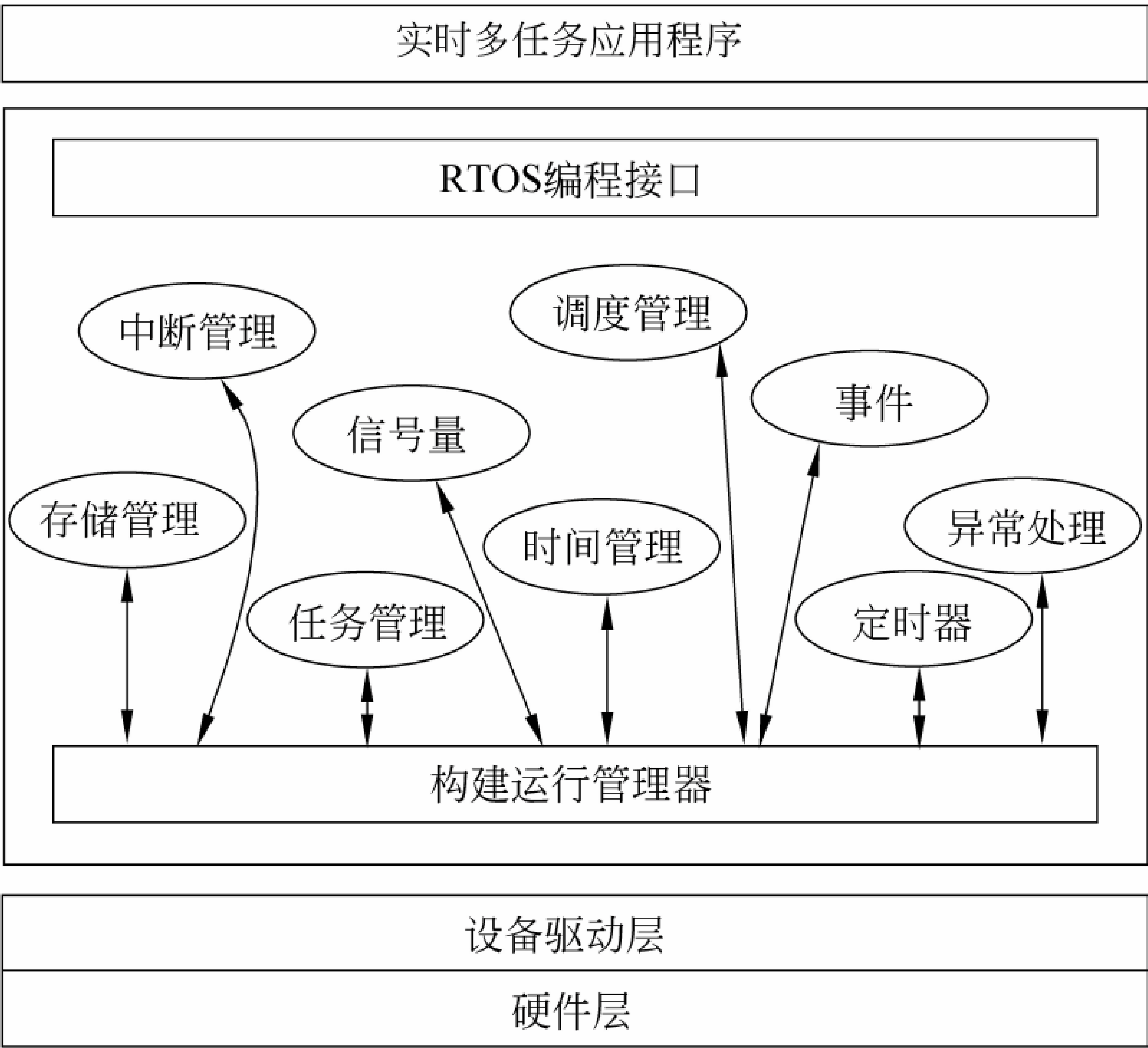



图 12.4　嵌入式操作系统的构件化结构

构件化结构具有以下特点：

（1）所有嵌入式操作系统抽象都由可加载的构件实现，配置灵活，裁剪方便。

（2）构件之间具有统一标准的交互式界面，便于用户掌握，方便应用程序开发。

（3）传统服务作为一个构件或由一些相互协作的构件构成，可为应用软件开发提供统一的编程接口。

（4）通过构件组装检验，可以确保生成的嵌入式操作系统满足设计约束。

（5）可以提供硬件无关性支持。

构件化结构应用在 TinyOS 等嵌入式操作系统中。

总而言之，上述的体系结构都是目前较为流行的结构，从中充分体现了嵌入式操作系统的高实时性、高可靠性和可裁剪性等特点。

# 12.2 嵌入式操作系统的任务管理

在所有的嵌入式操作系统中，其中重要的不可或缺的部分就是任务管理。实时多任务管理是嵌入式实时操作系统的核心和灵魂，决定了操作系统的实时性能。它通常包含优先级设置、多任务调度机制和时间确定性等部分。

## 12.2.1 多任务机制

**1. 任务**

任务（task）是嵌入式操作系统中最重要的操作对象，它是指拥有所有 CPU 资源的简单程序。在进行实时应用设计时，嵌入式操作系统通常将工作分割成多个任务，每个任务处理一部分问题，并被赋予一定的优先级、一套自己的 CPU 寄存器及堆栈。

随着应用的复杂化，一个嵌入式系统可能要同时控制/监视多个外设，要求有实时响应，有很多处理任务，各个任务之间有多种信息传递，因此，在嵌入式操作系统中，对应用需求需要采用多任务处理机制，以分时方式运行多个任务，并对其进行任务管理。

任务根据对截止时间的要求可以分为硬实时任务和软实时任务。硬实时任务是指需要得到及时执行的任务，如果在规定的截止时间内未满足，就会出现严重后果；软实时任务是指那些在规定截止时间内未满足不会出现严重后果的实时性任务。在目前实际运用的嵌入式系统中，通常允许软硬两种实时性任务同时存在，其中一些任务没有时限要求，另外一些任务的时限要求是软实时的，而对系统产生关键影响的那些事件的时限要求则是硬实时的。

任务也可以根据执行是否呈现周期性来划分，可分为周期性任务和非周期性任务。周期性任务每隔一个固定的时间间隔就会执行一次，非周期性任务执行的间隔时间则为不确定的。

**2. 任务控制块**

任务控制块是多任务机制中任务存在的唯一标识。在用户提交任务后，任务管理为任务分配内存空间，将任务代码载入到空间内，分配系统资源，并为之创建任务控制块（Task Control Block，TCB）。一般来说，TCB 保存了与任务相关的如下信息：

（1）任务 ID，标识任务的唯一序号。

（2）任务状态，任务的当前状态，如就绪、运行等。

（3）任务类型，任务的类型，包括硬实时任务、软实时任务和后台任务等。

（4）任务优先级，标识任务的优先级别，描述任务执行的优先度。

(5) 任务上下文指针，指向保存任务上下文的位置。

(6) 任务存储器指针，指向任务代码存储器、数据存储器和堆栈的指针。

(7) 任务系统资源指针，任务使用的指向系统资源(包括信号量等)的指针。

(8) 任务指针，指向其他 TCB 的指针。

(9) 其他参数，其他相关任务参数。

每个任务的任务控制块采用链表的方式进行存储，根据任务的状态，分为多种任务控制块链表。

**3. 任务上下文切换**

任务上下文是指任务运行的环境。例如，针对 x86 的 CPU，任务上下文可包括程序计数器、堆栈指针和通用寄存器的内容。

任务上下文管理负责嵌入式操作系统内核中任务管理部分对任务寄存器上下文的创建、删除以及切换等操作。任务的寄存器上下文是操作系统内核所管理的任务的重要组成部分，是 CPU 内核的寄存器中内容的映像，因此上下文管理的实现依赖于 CPU 内核中寄存器的组织，是与体系结构密切相关的。通用硬件抽象层的任务上下文管理统一定义体系结构中的寄存器上下文的保护格式，提供了任务管理对任务上下文的基本操作的 API 接口。

多任务系统中，上下文切换是指 CPU 的控制权由运行任务转移到另外一个就绪任务时所发生的事件，当前运行任务转为就绪(或者挂起、删除)状态，另一个被选定的就绪任务成为当前任务。上下文切换包括以下 6 个基本步骤：保存当前任务的运行环境；更新当前运行任务的状态；移动当前任务到相应队列；调度另一个任务；更改其状态为运行；恢复将要运行任务的上下文运行环境。上下文的内容依赖于具体的 CPU，而且与调度的过程有关。

**4. 任务控制**

任务控制主要是完成对任务状态的直接控制与访问。通常任务控制是通过系统调用来体现的，主要包括创建任务、删除任务、挂起任务、唤醒任务、设置任务属性和改变任务优先级等内容，这些控制一般采用原语的形式实现。

创建任务的过程是分配任务控制块的过程。在创建任务时，通常需要确定任务的名字和任务的优先级等内容，确定任务所能使用的堆栈区域，在任务创建成功后，通常会向用户返回一个标识该任务的 ID，以实现对任务的引用管理。

删除任务是指将任务从系统中去掉，释放对应任务控制块的过程。

挂起任务是把任务变为等待状态，唤醒任务操作则是把任务转换为就绪状态。

设置任务属性可以用来设置任务的抢占、时间片等特性，以确定是否允许任务在执行过程中被抢占或是对同优先级任务采用时间片轮转方式运行等。

改变任务优先级用来根据需要改变任务的当前优先级。

通过获取任务信息可以获得任务的当前优先级、任务属性、任务名字、任务上下文和任务状态等内容，便于用户进行决策。

### 12.2.2 任务状态和任务状态迁移

由于多任务机制管理中，任务要参与资源的竞争，只有在所需资源得到满足的情况下才能得到执行。然而，任务拥有的资源情况是不断变化的，这将导致任务状态也表现出不断变

化的特性。不同的实时内核实现方式对状态的定义不尽相同，但都包括以下 3 种基本状态：

（1）等待，任务在等待 I/O 完成或者等待某事件的发生。

（2）就绪，任务已经得到需要运行的资源，并等待获得处理器资源。

（3）执行，任务获得处理器和其他所有需要的资源，相关代码正在被运行。

在单处理器系统中，任何时候只有一个任务处于运行态。如果没有任何任务需要运行，那么内核会运行一个空闲任务。任何一个可以执行的任务都必须处于就绪态，实时内核会从所有就绪的任务中使用合适的调度策略选择一个运行。当一个任务请求 I/O 操作或者等待信号量时将会处于等待态。

在一定条件下，任务会在不同的状态之间进行转化，称为任务状态迁移。任务状态迁移情况如图 12.5 所示。对于处于就绪态的任务，获得 CPU 后，处于运行态；处于运行态的任务如果被高优先级任务所抢占，或者执行的时间片期满，任务又回到就绪态；运行的任务如果需要等待某些资源，任务被切换到等待态；对于等待态的任务，如果等待的资源或事件满足，就会转换为就绪态，等待被调度执行。

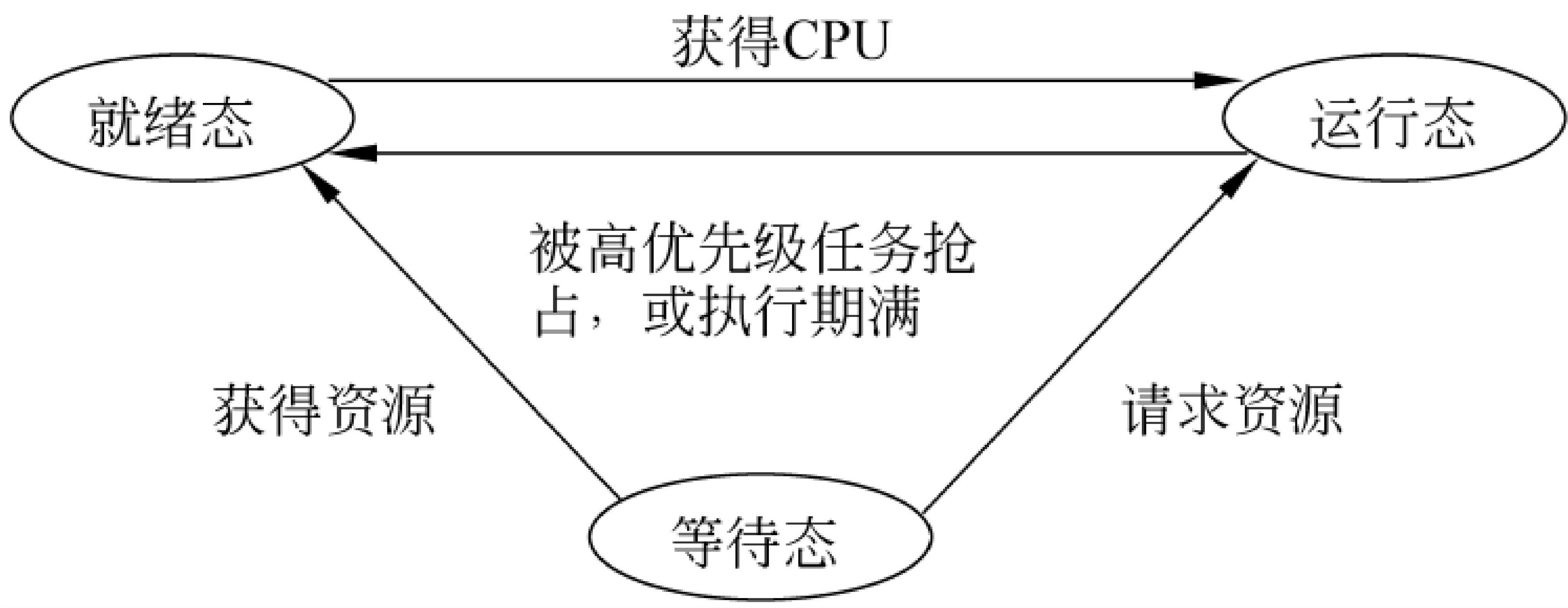



图 12.5　任务状态迁移

图 12.6 描述了 3 个任务状态的迁移过程。图中包含 3 个任务和一个调度程序。调度程序确定下一个需要投入运行的任务，因此调度程序本身也占用一定的处理时间。

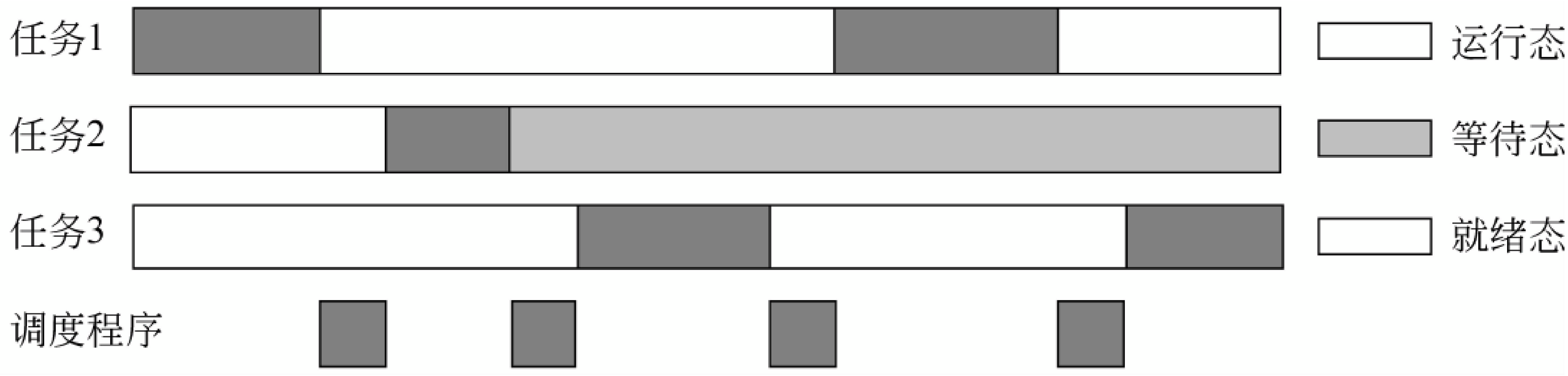



图 12.6　任务状态迁移示意

## 12.2.3　任务调度

调度是内核的主要职责之一，任务调度用来确定多任务环境下任务执行的顺序和在获得 CPU 资源后能够执行的时间长度。

任务调度所要解决的问题包括 3 个部分：第一，确定按什么原则分配 CPU 资源，即任务调度算法；第二，确定何时分配 CPU 资源，即任务调度的时机；第三，如何分配 CPU 资源，即任务调度的过程。

实时操作系统中大都采用抢占式调度算法，这样，硬实时任务能够打断软实时任务的执行，从而确保硬实时任务的截止时间能够得到满足。常见的调度算法包括基于优先级的抢

占式调度算法和时间片轮转调度算法等。

**1. 基于优先级的抢占式调度算法**

基于优先级的抢占式调度算法的基本思想是，具有最高优先级的任务先执行。每个任务都赋予优先级。任务越重要，赋予的优先级就越高。当就绪队列中有一个任务具有比当前执行任务更高的优先级时，实时内核进行任务切换，将当前任务停止执行，保存其上下文，并把 CPU 的控制权交给更高优先级的任务，使其运行。

优先级的分配方式可分为静态分配和动态分配两种。静态优先级是指应用程序执行过程中诸任务的优先级不变。在静态优先级系统中，各个任务以及它们的时间约束在程序编译时是已知的。动态优先级指应用程序执行过程中，任务的优先级是可变的。

**2. 时间片轮转调度算法**

当两个或两个以上的任务具有相同的优先级，且这些就绪任务是优先级最高的任务时，内核按照这些任务就绪的先后次序调度，允许一个任务运行事先确定的一段时间，然后又调度第二个任务执行确定的一段时间，依次类推地循环，这种调度算法为时间片轮转算法。其中，每个任务所执行的一段时间为时间片。内核在满足以下条件时，把 CPU 控制权交给下一个就绪态的任务：

(1) 当前任务运行的时间片到期。

(2) 当前任务在时间片还没结束时已经完成了。

多数实时内核采用基于优先级调度的算法，但有些实时内核常常会实现混合调度方案。例如，在 Linux 系统中，调度算法最基本的一类是基于优先级的调度，优先级高的任务先运行，而相同优先级的任务按照轮转方式进行调度，另外，Linux 也实现了基于动态优先级的调度方法。一开始，利用静态优先级的方法设置任务的优先级，然而它允许调度程序根据需要来提升或降低优先级。

当调度程序决定新的任务获得 CPU 的使用权时，这时内核将执行任务切换。任务切换过程为：首先保存当前任务的上下文，即 CPU 寄存器中的全部内容。这些内容可以保存在任务的自己的栈中，也可以保存在 TCB 中。然后，将需要运行的任务的上下文从该任务的栈中重新装入 CPU 的寄存器，并开始运行。任务切换过程增加了应用程序的额外负荷。CPU 的内部寄存器越多，额外负荷就越重。

### 12.2.4 任务间通信

由于实时操作系统中存在任务的调度，各个任务之间不可避免地存在相互协同的关系，这种协同就是任务间的通信。一般嵌入式操作系统都会提供各种任务间通信的方法，包括：

(1) 共享内存机制，数据的简单共享。

(2) 信号量机制，基本的互斥与同步。

(3) 消息队列与管道，同一 CPU 内多任务间消息传递。

在嵌入式多任务系统中，嵌入式实时多任务应用是由多个任务、多个中断处理过程以及嵌入式实时操作系统组成的有机整体。因此，各种任务之间存在着不同的关系，它们之间必须协调工作、相互配合，彼此交换信息。

**1. 共享内存机制**

共享内存机制是任务之间最直接、最明显的通信方法，即不同的任务通过访问同一地址

空间来完成任务之间信息的交换。如图 12.7 所示。由于大部分嵌入式系统的任务共存于单一的线性地址空间，在多个任务间共享数据结构非常容易，只要通信双方采用协商一致的数据结构即可。这些一致的数据结构包括各种类型的全局变量、双向链表和环形队列等复杂的数据结构。




图 12.7　共享内存机制示意图

当某一地址空间用于数据交换时，为了避免冲突，对于内存的锁定是非常重要的。两个或多个任务读写某些共享数据时，最后的结果取决于任务运行的精确时序，有可能得到错误值，这样必须以某种手段确保当一个任务在使用一个共享变量或文件时，其他任务不能做同样的操作。锁定内存主要有关中断、抢占禁止和用信号量锁定资源等方法。一般来说，关中断是最有效的解决互斥的方法。但这种方法对于实时应用来说，由于阻止了系统对外部事件的响应，无法满足实时性的要求。同样，中断延迟也是不能接受的。

**2. 信号量机制**

信号量机制是任务间少量信息的通信、提供同步和互斥的主要手段。通过信号量机制可以对共享资源上锁，从而限制对共享资源的同时操作，协调任务的执行。针对不同类型的问题，在嵌入式实时操作系统中可以使用以下几种信号量：

（1）二进制信号量，用于解决同步问题的最常用信号量。

（2）互斥信号量，为解决具有内在互斥问题、优先级继承、删除安全和递归等情况而最优化的特殊的二进制信号量。

（3）计数信号量，用于解决一种资源的多个实例需要保护的信号量。

在嵌入式实时系统中，这些信号量将根据用途而在具体实现中做专门处理，以提高程序的执行效率和可靠性。

**3. 消息队列与管道**

信号量机制提供的信息量是非常有限的，通常只是作为解决通信中涉及的互斥和同步问题的底层机制。而消息队列作为一种更高级的通信方式，能够处理同一处理器的各个任务间传递任意长度（理论上只受物理内存和机器字长限制）的信息。消息队列是一个类似于缓冲区的对象，通过消息队列，任务之间发送和接收消息，实现带数据的通信和同步。如图 12.8 所示，任务 1 和任务 2 之间采用两个消息队列来完成双工通信。

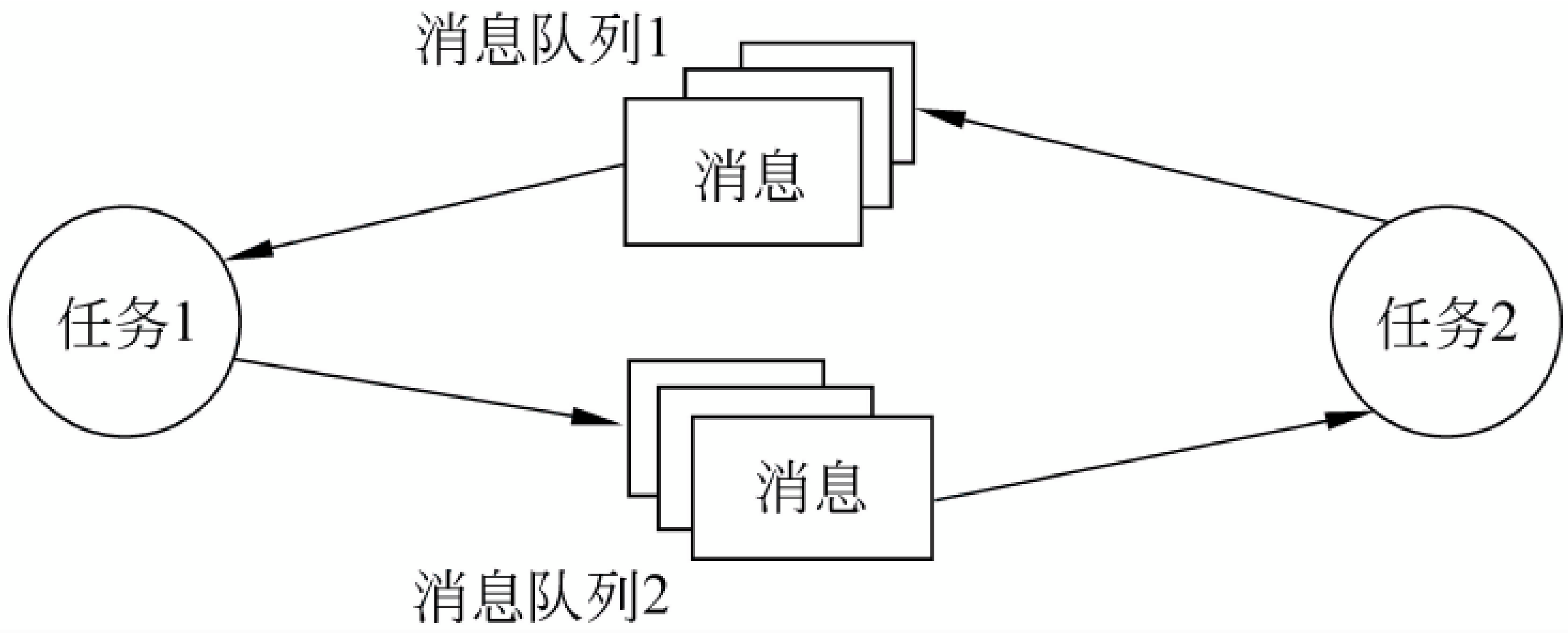



图 12.8　任务间消息队列通信示意图

图 12.8 中任务与消息队列之间是单向的，任务 1 和任务 2 分别发送消息到消息队列 2

和消息队列1，而两者分别从消息队列1和消息队列2中检索消息。

在客户/服务器模型中，一般采用多个消息队列来维护通信。服务器创建请求队列，并侦听队列上的客户请求，客户请求将放入不同的请求队列，并在各自创建的响应队列上取回服务器响应数据，如图12.9所示。

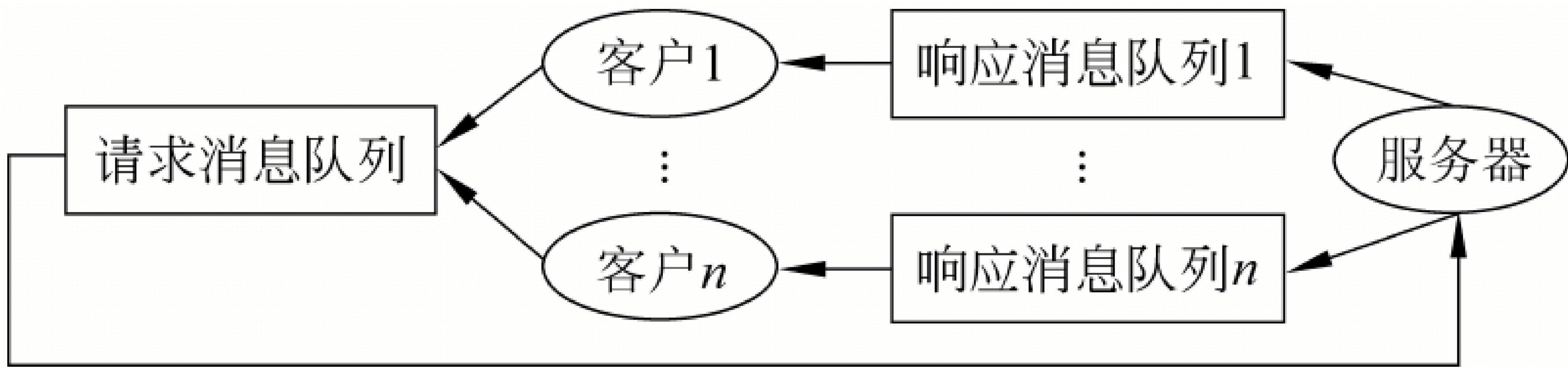



图12.9 客户/服务器消息队列通信示意图

在嵌入式实时操作系统中，应用消息队列将要更多地考虑优先级。消息队列的优先级主要是消息自身的优先级和阻塞任务队列的优先级。消息自身优先级决定了消息队列中多条消息提交给接收任务的顺序，这个优先级对消息接收者是透明的，接收者总是接收链队列首部的消息；阻塞任务队列优先级决定阻塞任务队列中多个发送者任务或多个接收者任务谁先被执行的顺序。一个消息队列包含了发送者阻塞队列和接收者阻塞队列，对于同一个消息队列而言，两个优先级是一致的。

管道是UNIX操作系统中传统的进程通信技术，它主要实现单向数据交换，在管道创立时返回两个端口的两个描述符，数据通过一个描述符写，通过另一个描述符读，数据在管道内像一个非结构字节流，按FIFO的次序从管道中读出。

与消息队列不一样的是，管道不存储消息，在管道中流动的数据也不是结构化数据，而是由字节流组成的。另外，管道中的数据不区分优先级，数据流严格地按照先进先出的方式执行。

在嵌入式实时操作系统中，管道由于其简单灵活而被应用。因为管道支持select()，任务可以同时等待包括管道在内的一系列I/O设备上的数据，而且管道提供的几个I/O控制命令也比较有用。

### 12.2.5 VxWorks任务管理

VxWorks是美国WindRiver公司设计开发的一种实时多任务的嵌入式操作系统，它由400多个相对独立的、短小精炼的目标模块组成，以其良好的持续能力、高性能的内核、友好的用户开发环境和卓越的实时性被广泛地应用于通信、军事、航空和航天等高精尖技术及实时性要求极高的领域中，如卫星通信、军事演习、弹道制导和飞机导航等。用户可根据需要选择适当的模块来裁剪和配置系统，系统的链接器也可按应用的需要自动链接一些目标模块，这样，通过目标模块之间的按需组合，可得到许多满足功能需求的应用。本节将从任务的状态、调度算法及任务之间的通信这几方面介绍VxWorks的任务管理功能。

**1. 任务控制块**

VxWorks中任务是竞争系统资源的最小运行单位。在VxWorks内核中，任务能快速共享系统的绝大部分资源，同时有独立的上下文来控制个别任务的执行。

对于每一个任务，系统使用任务控制块(TCB)来描述任务，每一个任务与一个TCB关联。TCB的组成如下：

(1) 任务的当前状态,描述任务当前所处的状态。

(2) 任务的优先级,采用 0～255 的数字表示,数字越小,优先级越高。

(3) 资源列表,描述任务所要等待的事件或资源。

(4) 任务程序码的起始地址,任务所存放的地址。

(5) 初始堆栈指针等。

另外,TCB 还被用来存放任务的上下文,任务的上下文保存的是一个任务进行切换时所要保存的当前系统状态相关的所有信息。

**2. 任务状态及状态转换**

VxWorks 中任务的基本状态分为 4 种,分别是就绪态(READY)、悬置态(PEND)、休眠态(SUSPEND)和延迟态(DELAY)。

(1) 就绪态:任务只等待系统分配 CPU 资源。

(2) 悬置态:任务需等待某些不可利用的资源而被阻塞。

(3) 休眠态:如果系统不需要某一个任务工作,则这个任务处于休眠状态。

(4) 延迟态:任务被延迟时所处的状态。

任务被创建以后进入挂起态,需要通过特定的操作使被创建的任务进入就绪态,这一操作执行速度很快,使应用程序能够提前创建任务,并以一种快捷的方式激活该任务。各种状态之间的转换如图 12.10 所示。

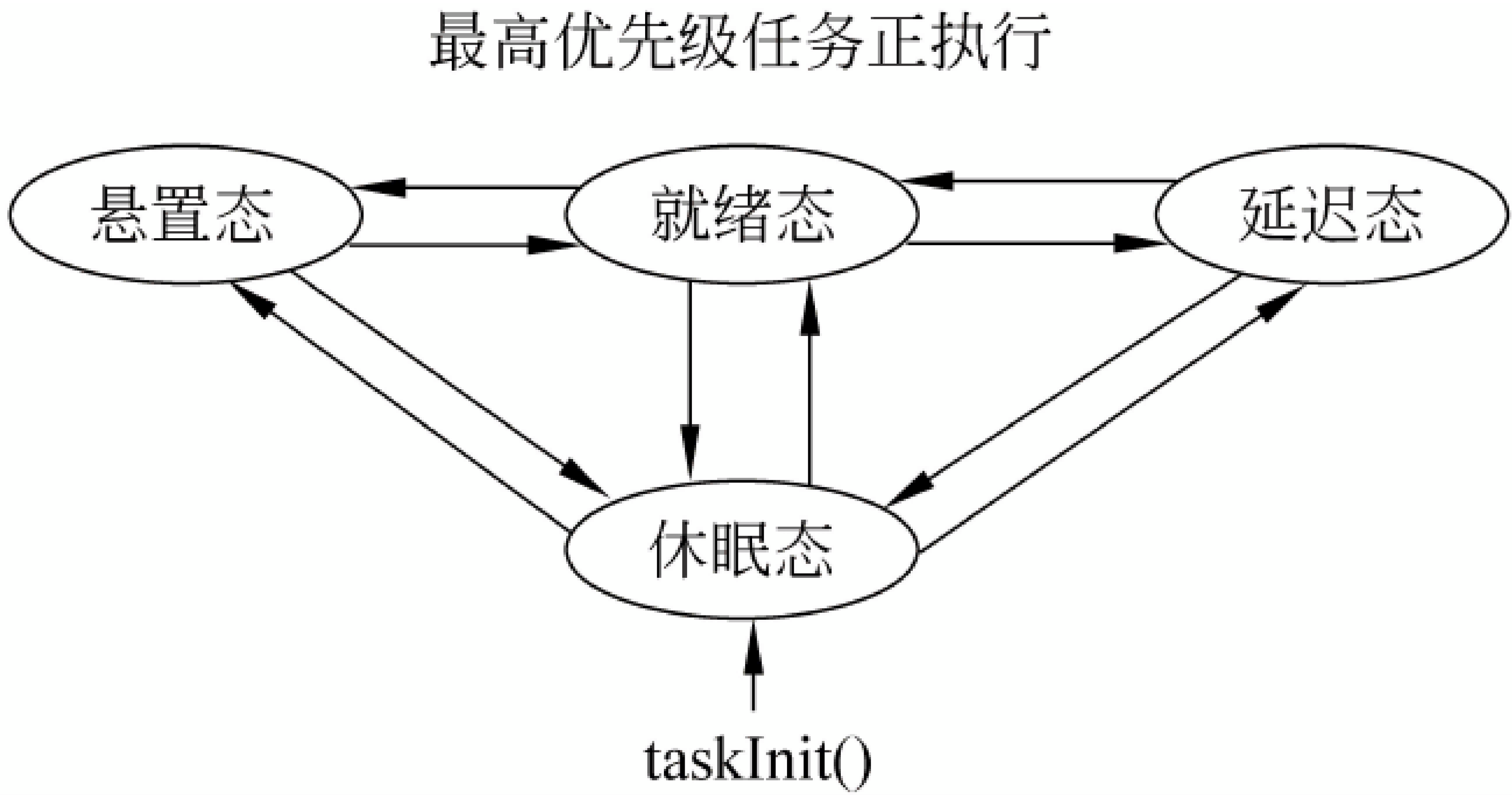



图 12.10 VxWorks 中的状态转换

VxWorks 中状态之间的转换通过一系列的函数调用来实现,如表 12.1 所示。

**表 12.1 VxWorks 有关状态转换的调用函数**

| 状 态 转 换 | 调 用 函 数 |
| --- | --- |
| 就绪态→悬置态 | semTake()/msgQReceive() |
| 就绪态→延迟态 | taskDelay() |
| 就绪态→休眠态 | taskSuspend() |
| 悬置态→就绪态 | semGive()/msgQSend() |
| 悬置态→休眠态 | taskSuspend() |
| 延迟态→就绪态 | expired delay |
| 延迟态→休眠态 | taskSuspend() |
| 休眠态→就绪态 | taskResume()/taskActivate() |
| 休眠态→悬置态 | taskResume() |
| 休眠态→延迟态 | taskResume() |

### 3. 任务调度

VxWorks 多任务的实现是由中断驱动的,即在每个系统时钟中断中实现任务的调度。VxWorks 的任务调度是采用基于抢占式优先级调度和时间片轮转调度混合的方法。

抢占式优先级调度算法是 VxWorks 中默认的算法,首先,每个任务有 256 个等级,采用 0~255 的数字表示,数字越小表示优先级越高。没有处于悬置态或休眠态的最高优先级任务将一直运行下去。当更高优先级的任务由就绪态进入运行时,该任务可以打断低优先级的任务而抢先执行,系统内核立即保存当前任务的上下文,切换到更高优先级的任务。只有在高优先级的任务执行完后,低优先级的任务才可以执行。

对于相同优先级的任务而言,系统采用时间片轮转调度算法。系统给处于就绪态的每个同优先级的任务分配一个相同的执行时间片。时间片的长度可由系统调用 KernelTimeSlice()通过输入参数值来指定。每个任务都有一个运行时间计数器,任务运行时每一时间滴答(tick)加 1。一个任务用完时间片之后,就进行任务切换,停止执行当前运行的任务,将它放入队列尾部,对运行时间计数器置零,并开始执行就绪队列中的下一个任务。当运行任务被更高优先级的任务抢占时,此任务的运行时间计数器被保存,直到该任务下次运行时。

另外,任务在执行的过程中可以调用 taskLock()函数来中止任务调度,从而保证该任务的执行不会被其他任务所打断,只有在调用 taskUnlock()后才会恢复任务调度。需要注意的是,如果调用 taskLock()的任务在调用 taskUnlock()之前又被挂起了(可能堵塞在信号量上等),那么系统就会搜索其他任务中优先级最高的那一个,并执行它,而在调用 taskLock()的任务恢复执行后,任务调度又会被中止。

当程序代码和数据出错时(如非法命令、总线或地址错误、被零除等),VxWorks 将进入异常处理,使引起异常的任务进入休眠态,保存任务在异常出错处的状态值。内核和其他任务继续执行。

### 4. VxWorks 任务管理函数

VxWorks 任务管理函数如表 12.2 所示。

**表 12.2　VxWorks 任务管理函数**

| 函　数　名 | 功　　能 |
|---|---|
| kernelTimeSlice() | 控制轮询式调度程序 |
| taskLock() | 取消任务的再调度 |
| taskUnlock() | 允许任务的再调度 |
| taskSpawn() | 生成(创建和激活)一个新任务 |
| taskCreate() | 创建一个新任务,但不激活它 |
| taskActivate() | 激活一个已经创建的任务 |
| taskSuspend() | 挂起一个任务 |
| taskResume() | 恢复挂起任务的执行 |
| taskRestart() | 重新开始一个任务的执行(即从头执行) |
| taskDelay() | 延时任务,延时单位是时间片 |
| taskIdSelf() | 得到调用任务的 id(正在运行的) |
| taskIdVerify() | 验证一个指定任务是否存在 |
| taskOptionsGet() | 获得用户自定义任务参数 |

| 函　数　名 | 功　　能 |
| --- | --- |
| taskOptionsSet() | 设置用户自定义任务参数 |
| taskIdListGet() | 将所有活动状态的任务 id 填写到个数组中 |
| taskInfoGet() | 得到一个任务的信息 |
| taskPriorityGet() | 获得任务的优先级 |
| taskPrioritySet() | 改变任务优先级 |
| taskRegsSet() | 设置一个任务的寄存器(但是不能被当前任务使用) |
| taskIsSuspended() | 检查一个任务是否在悬挂状态(suspended) |
| taskIsReady() | 检查一个任务是否准备运行就绪 |
| exit() | 结束正在运行的任务,释放内存 |
| taskDelete() | 结束制定的任务,释放内存 |
| taskSafe() | 保护当前任务,防止被删除 |
| taskUnsafe() | 取消 taskSafe()操作,即能够删除当前任务 |
| nanosleep() | 延时任务,延时单位是时间片 |

#### 5. 同步与通信

VxWorks 中的同步与通信的实现采用了各种机制,包括信号量、消息队列、管道和信号等。表 12.3 给出了 VxWorks 中所使用的各种同步与通信机制采用的函数。

**表 12.3　同步与通信机制相关函数表**

| 函　数　名 | 函 数 说 明 |
| --- | --- |
| semTake() | 获取一个信号量 |
| semGive() | 给出信号量 |
| semFlush() | 解锁所有正等待某一信号量的任务 |
| semBCreate() | 创建(产生并激活)一个二进制信号量 |
| semMCreate() | 创建(产生并激活)一个互斥信号量 |
| semCCreate() | 创建(产生并激活)一个计数信号量 |
| semDelete() | 中止并删除信号量 |
| msgQCreate() | 创建(产生并激活)消息队列 |
| msgQDelete() | 中止并删除消息队列 |
| msgQSend() | 向消息队列发送消息 |
| msgQReceive() | 从消息队列接收消息 |
| sigvec() | 用户可通过该调用把自己的处理函数连接在特定的异常处理信号上 |

## 12.3　内 存 管 理

内存管理机制是嵌入式系统中一个重要的内容,它必须满足实时性、可靠性和高效性的需求,因此,在嵌入式实时操作系统中,通常需要根据系统的要求采用特定的内存管理。

大体上来说,嵌入式系统所用到的内存管理机制包括虚拟内存管理机制和非虚拟内存管理机制。采用虚拟内存管理机制的嵌入式系统中含有 MMU,可以用于完成虚地址到物理地址的转换,从而使系统可以运行体积比物理内存还要大的应用程序。而在许多小型嵌入式系统中并未实现虚拟内存管理机制,常见的非虚拟内存管理机制是动态分区内存管理

机制，可以满足多道程序设计的需求，允许多个用户程序共享内存空间。

### 12.3.1 动态内存管理机制

动态内存管理机制主要管理空闲内存块，采用最佳适应算法和首次适应算法进行内存分配。最佳适应算法将所有的空闲内存块由小到大链接，首次适应算法中的空闲内存块是按地址由小到大进行链接的。在进程提出申请的时候，系统将从空闲块的首地址开始查找，找到满足需求的内存后，返回分配起始地址。

动态内存分配会导致响应和执行时间不确定、内存外碎片等问题，但是它的实现机制灵活，给程序实现带来极大的方便，有的应用环境中动态内存分配甚至是必不可少的。比如，嵌入式系统中使用的网络协议栈，在特定的平台下，为了比较灵活地调整系统的功能，在系统中各个功能之间作出权衡，必须支持动态内存分配。

大多数的系统是硬实时系统和软实时系统的综合，也就是说，系统中的一部分任务有严格的时限要求，而另一部分只是要求完成得越快越好。按照 RMS(Rate Monotonous Scheduling)理论，这样的系统必须采用抢先式任务调度；而在这样的系统中，就可以采用动态内存分配来满足那些可靠性和实时性要求不那么高的任务。采用动态内存分配的好处就是给设计者很大的灵活性，可以方便地将原来运行于非嵌入式操作系统的程序移植到嵌入式系统中。大多数实时操作系统提供了动态内存分配接口，例如 malloc 和 free 函数。

### 12.3.2 VxWorks 动态内存管理函数

VxWorks 中也采用了动态内存管理机制，动态内存分配采用最先匹配(first-fit)算法，即从空闲链表中查找内存块，然后从高地址开始查找，当找到第一个满足分配请求的空闲内存块时，就分配所需内存并修改该空闲块的大小。空闲块的剩余部分仍然保留在空闲链表中。当从大的内存块中分配出新的内存块时，需要将新内存块头部的一块空间(称为“块头”)用来保存分配、回收和合并等操作的信息，因此，实际占用的内存大小是分配请求的内存大小与块头开销之和。分配函数 malloc()返回的地址是分配请求所获可用内存的起始地址，为优化性能，malloc()会自动对齐边界，分配的内存的起始地址和大小都是边界值的整数倍，因此有可能造成内部碎片。块头的开销与处理器的体系结构有关，对于 x86 体系结构来说，块头的开销是 8B，通常这部分会被自动以 4B 边界对齐，以减少出现碎片的可能并优化性能。

VxWorks 提供了多种分配和释放内存的函数以满足不同的要求。其中基本的分配与释放函数是 memPartAlloc()和 memPartFree()，其他函数是在这两个函数的基础上实现的，如表 12.4 所示。

**表 12.4 动态内存分配和释放函数**

| 函　数　名 | 所属库 | 功 能 描 述 |
|---|---|---|
| malloc() | memPartLib | 从系统内存分区中分配指定大小的内存，与 ANSI C 兼容 |
| free() | memPartLib | 释放由 malloc()或 calloc()分配的内存块 |
| calloc() | memLib | 为一组元素分配内存，与 ANSI C 兼容 |

| 函　数　名 | 所属库 | 功 能 描 述 |
|---|---|---|
| cfree() | memLib | 释放由 calloc()分配的内存块 |
| valloc() | memLib | 从系统内存分区中分配指定大小的内存，并确保所分配的内存是从页边界开始的 |
| realloc() | memLib | 为指定的块重新分配内存，新分配的内存可能不在原位置上，但可以保留原内存块中的内容，与 ANSI C 兼容 |
| memAlign() | memLib | 从系统内存分区中分配内存，并确保内存块的起始地址是指定 alignment 参数的整数倍 |
| memPartAlignedAlloc() | memPartLib | 从指定内存分区中分配内存，并确保内存块的起始地址是指定 alignment 参数的整数倍 |
| memPartRealloc() | memLib | 从指定的分区中重新分配内存，并保留原来内存块的内容 |
| memPartAlloc() | memPartLib | 从指定内存分区中分配内存 |
| memPartFree() | memPartLib | 释放指定分区中的内存 |

VxWorks 5.x 也允许用户建立并管理自己的内存分区(memory partition)，并提供相应的函数，如表 12.5 所示。

**表 12.5　内存分区操作函数**

| 函　数　名 | 所属库 | 功 能 描 述 |
|---|---|---|
| memPartCreate() | memPartLib | 创建内存分区 |
| memPartAddToPool() | memPartLib | 向指定内存分区中增加内存 |
| memPartAlignedAlloc() | memPartLib | 从指定分区中分配内存，并确保该内存块的起始地址在特定的边界上 |

每个分区有各自的分区 ID。ID 实际是指向分区内存管理数据结构 mem_part 的指针，定义在 memLib.h 中。

### 12.3.3　虚拟内存管理机制

有一些嵌入式处理器提供了 MMU，MMU 具备内存地址映射和寻址功能，它使操作系统的内存管理更加方便。如果存在 MMU，操作系统会使用它完成从虚拟地址到物理地址的转换，所有的应用程序只需要使用虚拟地址寻址数据。这种使用虚拟地址寻址整个系统的主存和辅存的方式在现代操作系统中被称为虚拟内存。MMU 便是实现虚拟内存的必要条件。

MMU 的主要功能是以存储页(memory page)为基本单元，将虚拟地址翻译为物理地址、提供内存访问权限和控制存储页 Cache 支持。其具体实现因体系结构而异，如图 12.11 所示为 MMU 在系统中所处的位置。

Linux 系统中实现的就是虚拟内存管理机制。在不同的体系结构下，使用了三级或者两级页式管理，利用 MMU 完成从虚拟地址到物理地址之间的转换，并采用“按需调页”策略，实现请求分页管理机制。

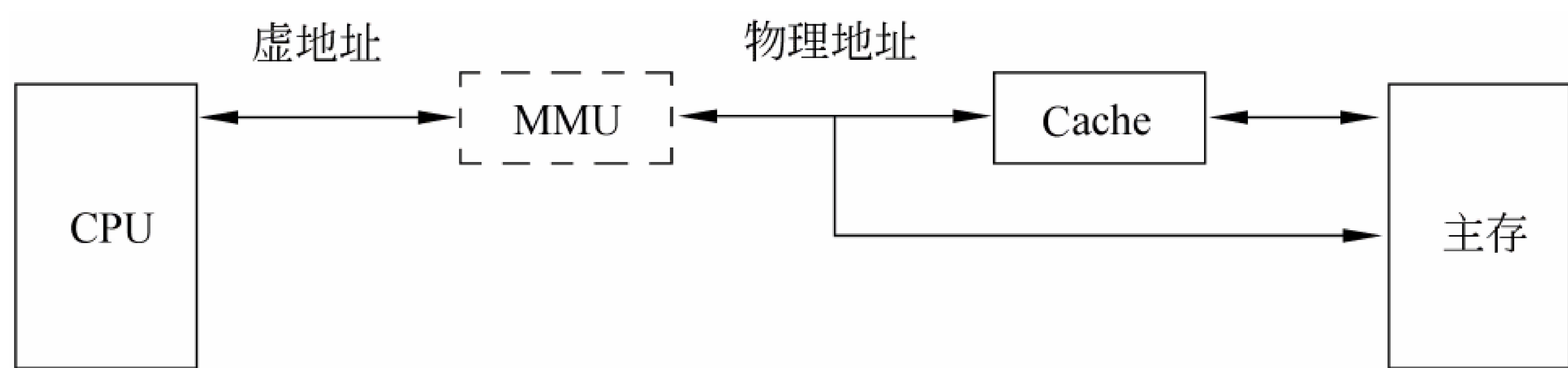



图 12.11　MMU 的位置

基于虚拟内存管理的内存的最大好处是：由于不同进程有自己单独的进程空间，十分有效地提高了系统可靠性和安全性。同时利用“按需调页”策略，既满足了程序的运行速度，又节约了物理内存空间。

## 12.3.4　VxWorks 虚拟内存管理

VxWorks 5.x 提供两级虚拟内存支持(virtual memory support)：基本级(basic level)和完整级(full level)。后者需要购买可选组件 VxVMI。

在 VxWorks 5.x 中有关虚拟内存的配置包括两个部分。第一部分是对 VxWorks 虚拟内支持级别和保护的配置，表 12.6 列出了这些选项。

表 12.6　VxWorks 虚拟内存配置常量

| 组　　件 | 描　　述 |
|---|---|
| INCLUDE_MMU_BASIC | 基本及虚拟内存支持，不需要组件 VxVMI |
| INCLUDE_MMU_FULL | 完整级虚拟内存支持，需要组件 VxVMI |
| INCLUDE_PROTECT_TEXT | 写保护 text 段，需要组件 VxVMI |
| INCLUDE_PROTECT_VEC_TABLE | 写保护异常向量表，需要组件 VxVMI |

以上配置一般在 BSP 的 config.h 中完成。

第二部分是内存页表的划分和属性配置。配置包括 BSP 的 config.h 中的 VM_PAGE_SIZE 和 sysLib.c 中的 sysPhysMemDesc。

VM_PAGE_SIZE 定义了 CPU 默认的页的大小，需参照 CPU 手册的 MMU 部分定义该值。

sysPhysMemDesc 用于初始化 MMU 的 TLB 表，它是以 PHYS_MEM_DESC 为元素的常量数组。PHYS_MEM_DESC 在 vmLib.h 中定义，用于部分内存的虚拟地址到物理地址的映射：

```
typedef struct phys_mem_desc
{
    void virtualAddr;                    /* 虚拟地址 */
    void *physicalAddr;                  /* 虚拟内存的物理地址 */
    UNIT len;                            /* 这部分内存的大小 */
    UNIT initialStateMask;
    UNIT initialState;                   /* 设置这部分内存的初始状态 */
}PHYS_MEM_DESC;
```

sysPhysMemDesc 中的值需要根据系统的实际配置进行修改。sysPhysMemDesc 中定

义的内存地址必须页对齐，且必须跨越完整的页。也就是说，PHYS_MEM_DESC 结构中的前 3 个域的值必须能被 VM_PAGE_SIZE 整除，否则会导致 VxWorks 初始化失败。

## 12.4　设备管理与文件系统

### 12.4.1　I/O 系统内部结构

嵌入式 I/O 系统主要由 I/O 设备、相关设备驱动程序以及一组标准的 I/O 操作函数组成。设备管理的主要任务如下：

（1）选择和分配 I/O 设备以便进行数据传输操作。

（2）控制 I/O 设备和 CPU（或内存）之间交换数据。

（3）为用户提供友好的透明接口，把用户和设备特性分开，使得用户在进行嵌入式应用程序开发时不必设计具体设备，由系统按用户的要求来对设备的工作进行控制；并需要为新增的用户设备提供与系统核心相连接的入口，以便用户开发新的设备管理程序。

（4）提高设备与设备之间、CPU 和设备之间以及进程与进程之间的并行操作程度。

由于用于嵌入式系统的输入输出设备很多，可从一个简单的串行通信设备到非常复杂的无线设备，各种设备只具有单一功能，而每个嵌入式系统中集成的设备不尽相同，因此，大多数嵌入式系统中将设备的管理分为两部分，一部分作为嵌入式操作系统内核的部分，另一部分则采用动态安装，在需要的时候在动态加载到嵌入式系统中。

嵌入式操作系统支持设备管理的分层结构，如图 12.12 所示。




图 12.12　I/O 系统的分层结构

I/O 系统的分层结构中，中断处理程序是整个设备管理的基础。当中断发生时，由中断处理程序执行相应的操作并解除相应进程的阻塞态，使其能够继续执行，设备请求队列中获得下一个设备驱动请求并驱动设备。平台相关设备层（Platform Dependent Device，PDD）是底层实现与操作系统的专用接口层，而逻辑设备驱动程序（Logical Device Driver，LDD）是操作系统的专用接口层，这两层构成了操作系统中的设备驱动程序。与用户相关的 I/O 软件是上层与用户的接口，是执行适用于所有设备的通用 I/O 功能，在操作系统中定义了一组标准的 I/O 操作函数，所有 I/O 设备驱动程序都支持这个函数集合。

操作系统对设备的管理分为 3 部分：

（1）缓冲技术，主要是用于提高数据的传输效率和安全，在外围设备与处理器进行数据传输时，通常将在主存开辟缓冲区来暂存 I/O 数据，经常采用单缓冲、双缓冲和多缓冲技术。

（2）设备的调度，主要是为等待各设备的多个进程安排使用的顺序。对于不同的设备，操作系统采用的调度策略不尽相同，但对于大多数设备而言，采用先来先服务是常见的策略，但对于磁盘而言，一般采用寻道时间最短调度策略来提高磁盘的效率。另外，设备调度的功能可位于操作系统的 I/O 管理模块，也可以下移到 I/O 处理程序中。

（3）设备的分配，即在对申请设备的进程进行设备分配时采用的策略。一般在分配时主要考虑分配的完整性和合理性，并要考虑对设备的控制器/通道等支持部件的分配。

### 12.4.2 实时内核的中断管理

嵌入式系统的中断处理是至关重要的，因为它是以中断方式通知系统外部事件的发生。中断是导致程序正常执行流程发生改变的事件。在实际应用中，中断可以分为硬件中断、自陷和异常3种类别。硬件中断又可分为可屏蔽中断和不可屏蔽中断，采用异步的方式执行；自陷和异常通常为软件中断，其处理程序以同步方式执行，处理的方式与硬件中断处理程序类似。

中断的处理过程分为中断检测、中断响应和中断处理3部分。中断检测在每条指令结束时进行，用于检测是否有中断请求或是否满足异常条件，因此，在指令周期中，一般采用中断周期来满足中断处理的需要，在中断周期中，处理器检查是否有中断发生，若有中断信号，则进入中断响应，对中断进行处理；若没有中断信号，处理器继续运行，并通过取指令周期提取当前程序的下一条指令。

中断响应是由处理器内部硬件完成的中断序列，其主要的工作是读出中断向量号，将标志位寄存器压栈，根据中断向量号查找中断向量表，并根据重量服务程序的首址转移到中断服务程序执行。

中断处理主要是执行中断服务程序，一般中断服务程序用于处理自陷、异常以及硬件中断，其处理过程主要分为以下几个方面：

（1）保存上下文，保存中断服务程序将要使用的所有寄存器的内容，以便退出中断服务程序之前进行恢复。

（2）若中断向量被多个设备共享，则轮询这些设备的中断状态寄存器，确定产生该中断信号的设备。

（3）获取中断相关的其他信息。

（4）对中断进行具体的处理。

（5）恢复保存的上下文。

（6）执行中断返回指令，使CPU的控制返回到被中断的程序继续执行。

在实时系统中，当对一个中断的处理还没有完成，又发生了另外一个中断时，系统将采用嵌套的中断处理方式，先执行高优先级中断的中断服务程序，然后才执行被打断的中断服务程序；而当在中断处理过程中，产生优先级比被中断程序更高的进程时，先执行优先级更高的进程，然后再执行优先级低的进程。

另外，在实时中断处理过程中，由于是嵌套中断处理，因此，在系统中必须考虑为任务栈预留足够的空间，从而使任务栈不溢出。

中断服务程序(ISR)运行在特定的空间，不同于其他任何任务，中断处理没有任务的上下文切换。所有的中断服务程序都使用同一中断堆栈，它在系统启动时就已根据具体的配置参数进行了分配和初始化。对于ISR的基本约束就是它们不能激活那些可能使调用程序阻塞的函数，例如，它不能获取信号量，如果该信号量不可利用，内核会试图让调用者切换到悬置态。然而，ISR能给出信号量。

一个ISR通常与一个或多个任务进行通信，有直接的也有间接的，作为输入输出事务

的一部分。这种通信的本质是驱动任务执行，从而处理中断和各种情况。这与任务到任务的通信和同步基本相同，但是有两点不同：

（1）一个 ISR 通常作为通信或同步的发起者，它通常返回一个信号量，向队列发送一个信息包或事件给一个任务。ISR 很少作为信息的接收者，它不可以等待接收信息包或事件。

（2）ISR 内的系统调用总是立即返回 ISR 本身。例如，即使 ISR 通过发送信息包唤醒了一个很高优先级的任务，它也首先必须返回 ISR。这是因为 ISR 必须先完成。

### 12.4.3　基本 I/O 操作流程

对于上层用户的 I/O 请求，系统将转化为具体的 I/O 要求，控制设备完成 I/O 操作。其基本的 I/O 操作流程主要分为以下几个步骤：

（1）将抽象要求转换为具体要求，即将 I/O 命令转换为控制器可接受的命令格式。

（2）检查 I/O 请求的合法性。

（3）读出和检查设备状态。

（4）传送必要的参数，如传输字节数、内存地址。

（5）设置工作方式，异步/同步通信。

（6）启动 I/O 设备，完成 I/O。

### 12.4.4　VxWorks 的 I/O 接口

VxWorks 中的 I/O 系统可以提供简单、统一、与设备无关的用户接口，由于 VxWorks 中的设备分为字符设备、随机存储块设备、虚拟设备、控制监视设备以及网络设备，因此这些接口对于每一种类型的设备也是不同的。根据操作方式的不同，VxWorks 操作系统中的 I/O 系统又分为基本 I/O 系统和缓冲 I/O 系统，操作系统为两种不同的 I/O 操作方式分别提供了不同的 C 语言函数库。基本 I/O 系统使用了与 UNIX 标准相兼容的 C 语言函数库，而缓冲 I/O 系统则使用了与 ANSIC 标准相兼容的 C 语言函数库。为了获得更快的速度和更好的兼容性，VxWorks 系统中的 I/O 都是经过优化设计的，而速度和兼容性对于 VxWorks 这样的实时系统来说都是至关重要的。

**1. 基本 I/O 接口**

基本 I/O 接口是 VxWorks 中最低级别的 I/O 接口。VxWorks 操作系统中的 I/O 操作与 I/O 原语相兼容，总共有 7 种基本 I/O 操作可以使用。

Create()　　创建并打开一个新文件

Open()　　打开一个新的或已存在的文件

Read()　　从文件中读取

Write()　　向文件中写入

Ioctl()　　其他功能

Close()　　关闭文件

Remove()　　移除文件

**2. 缓冲 I/O 接口**

为了进一步对设备提供支持并提高 I/O 系统效率，VxWorks 提供了对设备的缓冲操作。并且在操作方式上与基本 I/O 操作非常类似。

虽然 VxWorks 为基本 I/O 系统提供了很高的性能，但是每一次的 I/O 访问仍然需要一些与每个底层函数调用相关的额外开销。这些额外的开销可能会由两种原因引起。其一是为了实现某个操作，I/O 系统首先调用与设备无关的功能函数（比如说 read()或者是 write()等），然后再调用与驱动程序相关的底层函数，这种情况中虽然单次操作开销不大，但是如果频繁地访问就要花掉大量的开销。其二是大多数的驱动程序都会采用互斥或队列的机制来避免发生因同时调用一个驱动程序而导致多个任务互相影响的情况，这就会使系统频繁地将任务在各种状态之间切换，从而消耗掉了系统资源。

在实际中，单次执行这些操作所使用的开销是很小的。但是如果频繁地访问设备，例如每读取一个字节都使用一次 read()，就会给系统带来较大的开销，如下所示：

```
n=read( fd, &char, 1);
```

为了使上面例子的 I/O 操作更有效、更灵活，VxWorks 提供了缓冲机制，对大量可以缓冲的数据进行 I/O 读写操作。而这种缓冲对于用户是透明的。用户如果希望使用缓冲操作，只需要将文件用 fopen()函数以缓冲的方式打开就可以实现了。下面给出一个以缓冲方式打开文件的例子：

```
fp=fopen("/disk/myfile1" , "r");
```

fopen()函数所返回的是一个文件指针，它指向的结构包含已经打开的文件描述符、文件对应的缓冲区以及缓冲区的句柄。这个文件指针实际上是指向 FILE 类型的数据结构指针。基本 I/O 操作函数是通过文件描述符来确认一个文件，文件描述符是一个整型数；而缓冲 I/O 操作则是一个指向结构的指针，在这个指针里面包含相应文件的文件描述符。

**3. 其他 I/O 接口**

VxWorks 还提供了其他的一些格式化的 I/O 操作，包括 printf()、sprintf()、sscanf()、printer()、fdprintf()和信息记录函数 logMsg()。这些函数都是常用的格式化输入输出函数，符合 ANSIC 标准。

在其他常用的系统中 printf()、sprintf()和 sscanf()三个函数作为标准的 stdio 库中的一部分。但在 VxWorks 中却包含在 fioLib 函数库中。所以在使用这三个函数的时候可以不包含 stdio 库。虽然这些函数支持 ANSI 标准指定的功能，但 printf()却不是一个缓冲型的函数。如果希望执行带有缓冲功能的 printf()类型的输入输出函数，可以在应用程序中使用 fprintf()来代替 printf()。

VxWorks 在 fioLib 库中另外还提供了两种具有格式化操作但非缓冲类型的输出函数。printErr()函数在功能上与 printf()相似，不同的是。printErr()将格式化的字符串输入到标准错误输出设备，fdprintf()也提供类似的功能，它可以将格式化的字符串输出到任意文件中。

为了方便调试信息的输出以及在终端程序中输出调试信息，VxWorks 还提供在没有任务上下文或不对任务上下文进行 I/O 操作的情况下记录格式化信息的方式。信息的格式和参数将通过消息队列传递到执行记录的任务中，然后这个任务将信息格式化并输出。

### 12.4.5 文件系统架构及操作

嵌入式操作系统中的文件管理和通用的操作系统中的文件管理有所不同，其主要提供

文件存储、检索和更新等功能，但一般不提供保护和加密等安全机制。在某些情况下，文件系统还可以针对特殊的目的来进行定制，特别是随着特定应用的嵌入式操作系统的发展，文件系统的功能规整性、可伸缩性及其灵活性得到了很大的提高。

整个文件系统的架构如图 12.13 所示，自上而下包括 API 接口层、文件系统层、中间件层和物理驱动层。API 接口层是与用户的接口，通过 API 接口层为访问多个文件系统提供一个统一的抽象接口；文件系统层用于实现文件系统的初始化工作和与下层的格式化接口，是具体文件系统的操作接口及其他文件系统的可扩展接口；中间件层主要实现对存储设备的管理，包括向上的文件系统层接口、地址转换和物理设备驱动接口等功能；驱动层主要实现对不同存储介质的基本访问驱动接口，所有的嵌入式存储设备提供的接口在中间件层会进行封装，以实现具体设备之间的统一访问。



图 12.13　文件系统的分层结构

文件系统的操作包括对文件的操作和对目录的操作，所有的操作以系统调用和命令方式提供，具体操作如下：

（1）设置、修改对文件和目录的存取权限。

（2）建立、修改、改变和删除目录等服务。

（3）创建、打开、读写、关闭和撤销文件等服务。

嵌入式系统的文件系统通常支持几种标准的文件系统，如 FAT32、JFFS2 和 YAFFS 等。除支持标准的文件系统外，为提高实时性，有些嵌入式文件系统还支持自定义的实时文件系统，这些文件系统一般采用连续的方式存储文件。在嵌入式系统中，由于存储介质的不同，文件系统也有所不同，因此，一般嵌入式系统的文件系统是可裁剪、可配置的。可以根据嵌入式系统的要求选择所需的文件系统，选择所需的存储介质，配置可同时打开的最大文件数等；同时，考虑到存储设备的多样性及各种存储设备的不同特点，在设计嵌入式文件系统时，还需能方便地挂接不同存储设备的驱动程序，具有灵活的设备管理能力；并需考虑外部存储设备的性能、寿命等因素，发挥不同外存的优势，提高存储设备的可靠性和使用寿命。

### 12.4.6　VxWorks 文件系统

VxWorks 提供对字符设备和块设备的访问，其快速文件系统适合实时系统应用。它包括几种支持使用块设备(如磁盘)的本地文件系统。这些设备都使用一个标准的接口，从而使得文件系统能够被灵活地在设备驱动程序上移植。

VxWorks 也支持 SCSI 磁带设备的本地文件系统。VxWorks I/O 体系结构甚至还支持在一个单独的 VxWorks 系统上同时并存几个不同的文件系统。

VxWorks 支持 4 种文件系统，分别是 dosFs、rt11Fs、rawFs 和 tapeFs。

（1）dosFs：适用于块存取设备(磁盘)的实时操作，是与 MS-DOS 文件系统兼容的文件系统。

（2）rt11Fs：提供了一种简单的原始文件系统。该文件系统将整个磁盘当作一个单独的大文件。

（3）rawFs：适用于不使用标准文件或目录结构的磁带设备。实际上将磁带盘当作一

个原始设备并将整个磁带盘当作一个大文件。

(4) tapeFs：允许应用程序从按照 ISO 9660 标准文件系统格式化的 CD-ROM 设备上读取数据。

(5) TrueFFS：简称 TFFS，是为各种 Flash 存储器提供通用的块设备接口，支持 NOR 和 NAND 的 Flash 存储器。该文件系统将 Flash 存储器映射为一系列连续的块，并使用 block-to-flash 转换系统，方便地将块编号直接转换为 flash 地址。

另一方面，普通数据文件和外部设备都统一作为文件处理。它们在用户面前有相同的语法定义，使用相同的保护机制，这样既简化了系统设计，又便于用户使用。

VxWorks 中用户读写文件的接口是标准接口，处于最上层的应用对文件系统的操作接口包括 ioLib、ansiStdio、fioLib、dirLib 以及 usrFsLib。

ioLib 提供的是对文件读写的标准接口，包括 fopen()、fclose()、fread()、fwrite()及 fseek()等，在 ioLib 中还提供一些与文件系统相关的特殊接口，如 unlink()、rename()、lseek()、chdir()和 getwd()等。

dirLib 库提供的是目录文件列表读取的功能，它建立在 ioLib 之上，包括打开目录(opendir)、读取目录(readdir)、使目录读取位置从头开始(rewinddir)以及关闭目录(closedir)等操作，dirLib 还提供用于获取文件或文件系统状态信息的函数接口，如 fstat、stat 和 fstatfs 等。

usrFsLib 是在 ioLib 和 dirLib 之上的更实用的抽象，提供给用户熟悉的命令，如 ioHelp、cd、pwd、mkdir、rmdir、rm、chkdsk、ls、ll、lsr、llr、copy、cp、rename、mv、xcopy、xdelete、attrib、xattrib 和 diskFormat 等。

## 12.5 嵌入式操作系统的开发

当前，随着嵌入式软件的发展，越来越多功能强大的嵌入式操作系统也相继涌现，这些操作系统所采用的体系结构以及技术手段各有特色，在开发中具有以下方面的考虑。

**1. 主机/目标机的体系结构**

许多嵌入式操作系统在开发过程中都采取了主机/目标机的设计方法。主机/目标机的体系结构是将开发工具放在主机上，目标机上则放操作系统的核心模块，操作系统支持跟踪调试。如此，设计者便可在目标机上运行操作系统及应用软件，而开发和调试则通过主机进行，使开发的过程变得相对简单。

**2. 划分模块**

嵌入式系统在许多领域都有应用，不同的应用目的拥有不同的功能和结构，从而嵌入式操作系统也不同。因此，在设计嵌入式操作系统时，充分考虑系统的功能与结构的划分，这可增加嵌入式系统操作的适用性及灵活性。因此，在进行操作系统开发时，划分核心模块，将一些系统的核心功能独立出来，形成单独的可拆卸的模块，对于增强操作系统的模块性具有很大的意义，而且也成为嵌入式操作系统发展的一个总趋势。

**3. 充分利用现有资源**

虽然当今软件技术已高度发达，但从头设计一个操作系统也需要较长的时间，因此为减少操作系统开发的工作量，可充分利用现有的资源进行设计。

**4. 制定 API 标准**

为实现嵌入式操作系统的透明性以及无关性，方便用户的使用，就需要给用户提供标准以及实用的应用程序接口（API）。通过以上方式便可实现嵌入式操作系统。另外，在设计嵌入式操作系统时还需遵循以下原则：充分利用相关的计算机技术使得所设计的操作系统具有较高的性价比；要考虑到操作系统与嵌入式系统的匹配性；在满足嵌入系统功能的同时要尽可能简单，从而节约系统开销费用。

目前，嵌入式操作系统的开发一般和嵌入式软件的开发一起进行，需要一个交叉编译和调试环境，即编辑和编译软件在主机上进行（如在 PC 的 Windows 操作系统下），编译好的软件需要下载到目标机上运行（如在一个 PC 目标机上的 VxWorks 操作系统下），主机和目标机建立起通信连接，并传输调试命令和数据。由于主机和目标机往往运行着不同的操作系统，而且处理器的体系结构也彼此不同，这就增加了嵌入式开发的复杂性。

## 12.5.1 集成开发环境 Tornado

Tornado 是嵌入式实时领域里最新一代的开发调试环境，它给嵌入式系统开发人员提供了一个不受目标机资源限制的超级开发和调试环境，是为开发嵌入式操作系统（VxWorks）及其应用系统所提供的集成开发环境，Tornado Ⅱ中包含的工程管理软件可以将用户自己的代码与 VxWorks 的核心有效地组合起来，可以按用户的需要裁剪配置 VxWorks 内核；VxSim 原型仿真器可以让程序员在不用目标机的情况下直接开发系统原型，作出系统评估；功能强大的 CrossWind 调试器可以提供任务级和系统级的调试模式，可以进行多目标机的联调；优化分析工具可以帮助程序员从多种方式真正地观察、跟踪系统的运行，排除错误，优化性能。Tornado 的整体结构如图 12.14 所示，其包含 3 个高度集成的部分：

（1）运行在主机和目标机上的强有力的交叉开发工具和实用程序。

（2）运行在目标机上的高性能、可裁剪的实时操作系统 VxWorks。

（3）连接主机和目标机的多种通信方式，如以太网、串口线、ICE 或 ROM 仿真器等。

对于不同的目标机，Tornado 给开发者提供一个一致的图形接口和人机界面。当使用 Tornado 的开发人员转向新的目标机时，不必再花费时间学习或适应新的工具；对嵌入式应用开发者来说更重要的是，Tornado 所有的工具都是驻留在开发平台上的。在嵌入式系统工具发展历史上，Tornado 是第一个实现了当目标机资源有限时开发工具仍可使用而且功能齐全的开发环境。另外，所有工具都通过一个中央服务器（target server）处理与目标机的通信。所以无论连接方式是 Ethernet 还是串口线、ICE 仿真器、ROM 仿真器或客户设计的调试通道，所有工具均可使用。

Tornado Ⅱ 嵌入式开发系统的软件工具包含了核心工具（Core tools）和备选工具（Optional tools）两部分。

**1. 核心工具（Core tools）**

核心工具安装简单，通过紧密集成的开发环境可以在应用程序开发的任何阶段轻松地访问，而且不必关心目标系统的连接方式或目标系统的内存大小。

- 图形化的交叉调试器（Debugger）CrossWind/WDB：远程的源代码集成调试器，支持任务级和系统级调试，支持混合源代码和汇编代码显示，支持多目标机同时调试。

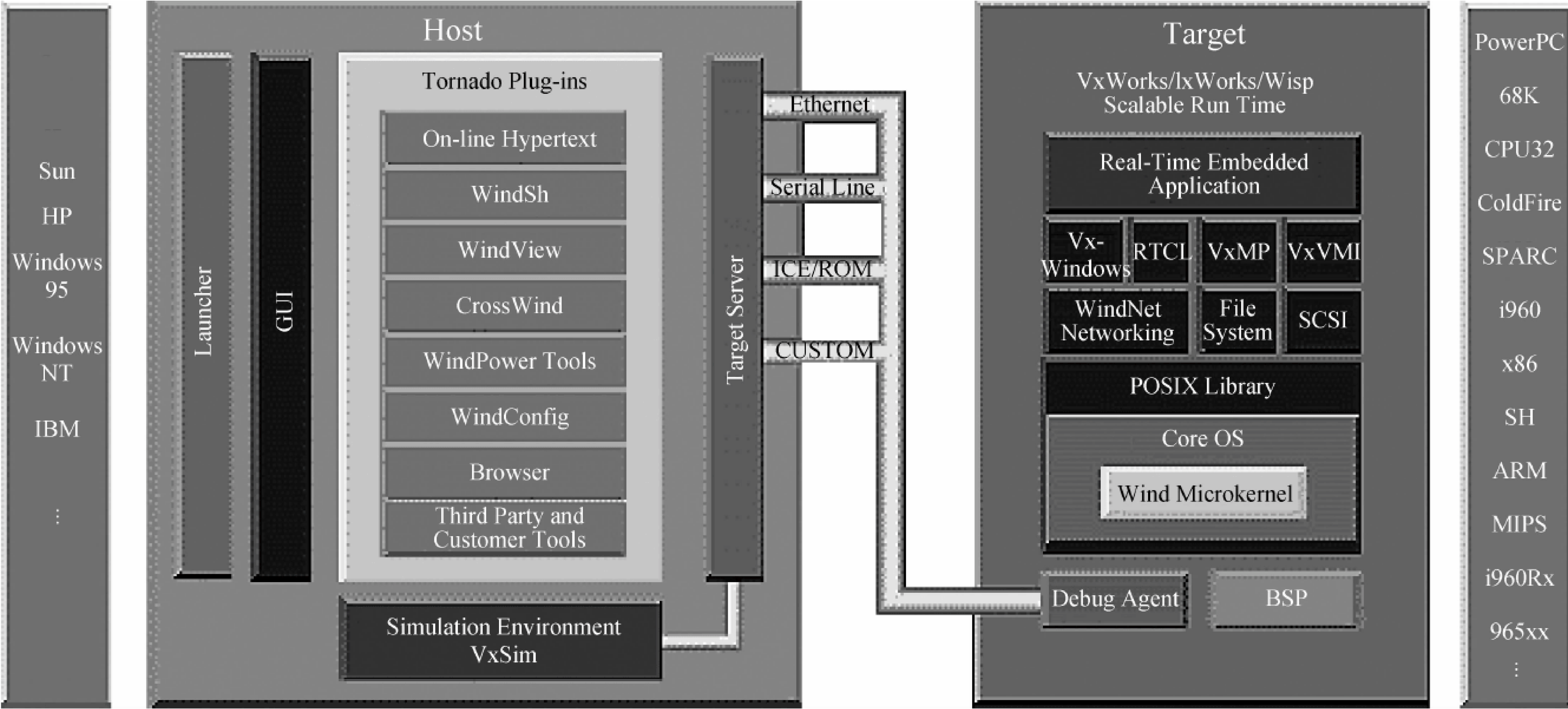



图 12.14　Tornado 整体结构示意图

开发者可以在目标运行系统上产生和调试任务，也可以将调试器和已经运行的任务连接在一起，这些任务可以是来自应用程序，也可以是来自任务级调试环境，具有很强的灵活性。

- 工程配置工具(Project Facility/Configuration)：强有力的图形化工具，提供对VxWorks操作系统及其组件进行自动配置的功能，可自动进行依赖性分析、代码容量计算、自动裁剪 wizard 以及自动生成维护 Makefile，单独的组件可以独立开发，并可以共享和重用。
- 集成仿真器(Integrated Simulator)：即 VxSim 仿真器，支持 CrossWind、WindView 和 Browser，提供与真实目标机一致的调试和仿真运行环境。VxSim 仿真器包含在各个软件包中，允许开发者可以在没有 BSP、操作系统配置和目标机硬件的情况下，使用 Tornado Ⅱ 迅速开始开发工作。
- 诊断分析工具(WindView for the Integrated Simulator)：图形化的动态诊断和分析工具，主要是向开发者提供目标机硬件上实际运行的应用程序的许多的详细情况，可图形化显示任务、中断和系统对象相互作用的复杂关系，并可以选择用于监视目标硬件系统行为的 WindView。
- C/C++ 编译环境(C/C++ Compilation Environment)：包含了交叉编译器、iostreams 类库和一些列的工具，用于支持 C 语言和 C++ 语言编程。交叉编译器包括 Diab C/C++ Compiler、GNU C/C++ Compiler 等；iostreams 类库支持 C++ 中的格式化的和类型安全的 I/O，可扩展到用户自定义数据类型，支持 C++ 应用程序开发的工业标准。另外，Tornado Ⅱ 还提供对 C++ 全面的支持，包括异常事件处理、标准模板库(Standard Template Library, STL)、运行类型识别(Run-Time Type Identification, RTTI)、支持静态构造器和析构器的加载器以及 C++ 调试器，从而保证工具与开发环境紧密地结合在一起。
- 主机目标机连接配置器(Launcher)：Launcher 允许开发者轻松地设置和配置一定的开发环境，并提供对开发环境的管理和许多管理功能。

- 目标机系统状态浏览器(Browser)：是 Tornado Ⅱ shell 的一个图形化组件，其主窗口提供目标系统的全面状态总结，允许开发者监视独立的目标系统对象：任务、信号灯、消息队列、内存分区、定时器、模块、变量，堆栈等。这些显示可根据开发者的选择进行周期性或条件性更新。
- 命令行执行工具(WindSh)：是 Tornado Ⅱ 所独有的功能强大的命令行解释器，可以直接解释执行 C 语言表达式，调用目标机上的 C 语言函数，访问系统符号表中登记的变量，还可以直接执行 TCL 语言。
- 多语言浏览器(WindNavigator)：用于提供源程序代码浏览，图形化显示函数调用关系，快速代码定位。
- 图形化核心配置工具(WindConfig)：Tornado Ⅱ的图形化核心配置工具(WindConfig)使用图形向导方式智能化地自动配置 VxWorks 内核及其组件参数。
- 增量加载器(Incremental Loader)：Tornado Ⅱ 的增量加载器(Incremental Loader)可以动态地加载新增模块并在目标机与内核实现动态链接运行，不必重新下载内核及未改动的模块，加快开发速度。

**2. 备选工具(Optional tools)**

备选工具包含以下项目：

- 软件逻辑分析仪 WindView：为开发者专门设计的符合实时嵌入式系统开发要求的高级的可视化工具。通过软件逻辑分析仪，开发者能看见嵌入式系统内部的动态运行过程，能以微秒级时间精度显示任务、中断服务程序和系统对象之间的复杂的相互关系，从而能让开发者迅速找到关键问题所在而忽略次要问题，理解特殊行为产生的原因，并找到解决问题的最佳方案。
- 原型仿真器 VxSim：是一个完整的原型和仿真工具，它可以帮助开发者在没有实际的目标硬件的前提下，先进行包括网络和基于多处理器的 VxWorks 应用程序设计和调试；同时，它还允许开发者在开发周期的前期就进行大量的应用程序测试，以降低更改软件的成本。
- 显示软件包 ScopePak：实时图形检测和数据收集工具，包括软件示波器 StethoScope 和跟踪示波器 TraceScope 两个工具。它允许开发者在应用程序运行时监视数据和函数调用(白盒监测工具)，观察系统中任何一组变量和内存的分配情况，发现可能被忽视的峰值和失灵，观察建立在特定事件上的触发集合；可以在程序运行时改变变量的值并将数据存盘；还可以看见代码改变、参数改变和外部事件的影响。显示软件包 ScopePak 是一个可以深入到应用程序内部的有价值的诊断工具，可提供程序的鲜活的分析报告而不影响实时性。
- 性能检测包 PerformancePak：是黑盒监测工具，包含 CPU 运行分析工具 ProfileScope 和内存使用分析工具 MemScope，不需要特殊的编译，可以提供 CPU 工作图和应用程序使用内存情况的详细分析图，可以指出系统缺乏效率的地方或需要开发者调整的地方，还可提供可视化工具在系统失败前控制内存使用，从而使开发的操作系统和应用程序获得最好的性能。
- 代码测试器 CodeTest：是白盒测试工具，包括 Converge Module 和 Memory Module 两部分。Converage Module 主要是确定代码中未经测试的部分和提供系统的动态

视图，Memory Module 主要是显示内存的动态分配情况，检查内存的泄漏。该工具的应用能让开发者获得最精确的测量，发现最难解决的跟踪动态内存分配问题。

### 12.5.2 VxWorks 的交叉编译开发环境

整个 VxWorks 操作系统的基本构成部件主要包括板级支持包 BSP（Board Support Package）、微内核 wind、进程管理、存储管理、设备管理、文件系统管理、网络协议及系统应用等几个部分，VxWorks 只占用了很小的存储空间，并可高度裁剪，保证了系统能以较高的效率运行。在 Tornado Ⅱ 系统中，为支持 WxWorks 操作系统的开发，同样也提供了 VxWorks 运行组件，包括板级支持包开发工具 BSP Developer's Kit、支持松散耦合分布式多处理器的 VxFusion、WindML & Zinc for VxWorks 图形界面开发包、Java 虚拟机以及 TrueFFS Flash 文件系统等几个部分。

**1. 板级支持包开发工具 BSP Developer's Kit**

板级支持包开发工具 BSP Developer's Kit 为嵌入式软件工程师开发自己的目标硬件的驱动程序，进而开发 BSP 方面提供了强有力的支持。板级支持包开发工具包括板级支持包开发工具基本包 BSP Developer's Kit Base Option 和板级支持包开发工具高级包 BSP Developer's Kit Value Option。

（1）板级支持包开发工具基本包 BSP Developer's Kit Base Option 主要面向的是并不绝对需要驱动程序代码样本的嵌入式开发者，其主要组成包括以下几部分：

- 板级支持包测试工具 BSP validation test suite：用来检查 BSP 和驱动程序的基本功能以及报告存在的问题。采用源代码形式呈现，可以运行在 Windows 95/98/NT、Solaris、SunOS 和 HP-UX 等操作系统的主机上。
- 板级支持包开发模板 Template BSP：针对各种 CPU（如 Motorola 68K/CPU32、Intel/AMD/Cyrixx86、MIPS、Morotorola/IBM PowerPC、SPARC 和 Intel i960 等），提供 BSP 开发的起点。
- 驱动程序开发模板 Template Driver：用于提供各种设备的驱动程序模板，包括串口（Serial）、以太网（Ethernet）、SCSI、中断控制器（Interrupt Controller）、VME、定时器（Timer）以及 Non-volatile RAM 等。
- SCSI 测试工具 SCSI Test Suite：用于 SCSI 的测试。
- 板级支持包开发工具文档 Documentation Set：描述板级支持包开发工具的帮助说明文档。

（2）板级支持包开发工具高级包 BSP Developer's Kit Value Option 主要面向的是愿意使用 WindRiver 公司提供的一般驱动程序源代码作为他们自己的驱动程序和 BSP 开发起点的开发者。板级支持包开发工具高级包 BSP Developer's Kit Value Option 包括几乎所有现成的标准驱动程序，例如 Ethernet、SCSI 等驱动程序源代码。其主要组成包括以下几部分：

- 板级支持包开发工具基本包 BSP Developer's Kit Base Option。
- Ethernet 驱动程序源代码 Ethernet Driver Source。
- SCSI 驱动程序源代码 SCSI Driver Source。

**2. 支持松散耦合分布式多处理器的 VxFusion**

VxFusion 是一个轻便的、独立于媒体的容错机制，这种机制建立在 VxWorks 的消息队列基础之上，主要用来开发分布式应用程序。

VxFusion 具有以下特点：

(1) 提供一个轻便的基于 VxWorks 消息队列的分布式机制。

(2) 本身独立于媒体，允许分布式系统无论以什么传输方式都可以实现有效的数据交换，而且用户不需要通信硬件。

(3) 通过在多节点系统中复制每个节点上已知对象的数据库来保证没有一对多的主从依赖关系所引起的每个单点的失效。

(4) 支持系统中点对点和广播方式的消息传递。

(5) 支持地点的透明性。即对象不用重写应用程序代码就可以实现在系统中无缝的移动。尤其特别的是，在多节点系统中不管对象的位置就可以实现向对象传递消息。

**3. WindML & Zinc for VxWorks 图形界面开发包**

WindML & Zinc for VxWorks 可以使开发者在 VxWorks 实时操作系统上以较低的系统开销设计实现丰富多彩的嵌入式 GUI。其只需要很小的内存就能获得高性能的图形输出，并容易移植，可裁剪，高度可定制，具有丰富的用户界面对象，支持单字节和双字节集以及通用的多媒体图形库，从而使其开发更丰富多彩。

**4. Java 虚拟机**

Java 虚拟机主要是 Personal Java for VxWorks，它能完全兼容 Personal Java，提供完整的基于 Java 线程的控制，并能使 VxWorks 任务和 Java 线程共同运行，相互同步，相互通信，具有设计一次就可以在任何地方运行、面向 Internet、安全性、可扩展性、可升级性、可裁剪性和可移植性等特点。

对于 Internet、Intranet 和消费电子的应用程序来说，Personal Java for VxWorks 与其他的 WindRiver 解决方案实现了无缝集成，其中包括 WindNet 网络产品。

**5. TrueFFS Flash 文件系统**

TrueFFS Flash 是 Tornado Ⅱ 开发环境中集成的快速闪存文件系统，它使用一系列嵌入式闪存设备来实现快速可靠的物理存储。在系统中采用仿真 VxWorks DOS 文件系统下的硬盘驱动器，开发者可以使用标准的文件系统界面来产生和操作一个文件系统，从而使闪存设备上的读写操作与在 DOS 文件系统设备上一样。

在实时应用系统的开发调试阶段，往往以 PC 作为主机来调试程序。一般在 Tornado Ⅱ 环境中，具体的软件调试环境搭建可以采用以下方法：

(1) ICE 方法。仿真头插在用户板的 CPU 上，image 在仿真器中运行，由仿真器的 CPU 在用户的硬件环境下运行来调试软件。其缺点是性价比不高。

(2) BDM 方式。主机通过并口连接的 Adapter 连接到 CPU 的 BDM 接口上，用户板上的 CPU 支持一种断点逻辑，通过这种硬件调试代理，进行 image 的调试。其缺点是 CPU 必须支持 BDM 方式。

(3) 串口调试方法。主机和目标机用串口线连接，在主机和目标机之间提供一种无缝连接。目标机需要使用 VxWorks 支持的网卡，image 通过以太网由主机下载到目标机。由于 VxWorks 支持的网卡比较常见，并且网络调试的速度很快，且支持网络功能，故网络连

接成为常用方法。

一般开发时常用后两种方法建立调试环境，具体方法由 config. h 中的 BOOT_LINE 决定。

### 12.5.3 实例开发的设计与实现过程

使用 Tornado 环境进行开发时，VxWorks 系统可以为用户提供大量的系统调用接口，针对开发工作和实时系统的特点，在进行实例开发的时候应当注意任务划分的合理性，要防止死锁、饥饿和优先级翻转，要考虑代码的可重入性、用户任务优先级以及访问资源的方法。

在进行开发前，首先要配置环境，具体包括以下几步：

(1) 配置模板联编文件。在创建程序之前，配置 VxWorks 模板联编文件 tornado. tmf 来指定使用 VxWorks 的环境信息，指定目标及目标机使用的 CPU 类型以及 Tornado 目标服务器工作所需要的主机名。CPU 类型必须与 VxWorks 中 BSP 板级支持包中对应的目标类型一致。同时，还需确定程序创建过程中所用到的 Tornado 工具的位置，包括编译器和目标模板等工具的位置。

(2) 创建程序。生成 Simulink 模块，创建和初始化 Simulink 的具体内容，初始化的内容包括模拟参数的设置、仿真结构的配置以及 Tornado 的配置，并创建应用程序，如图 12.15 所示。生成的应用程序的目标文件的扩展名为. lo(代表 loadable object)。初始化成功后，在模型目录下将生成一个. lo 文件。

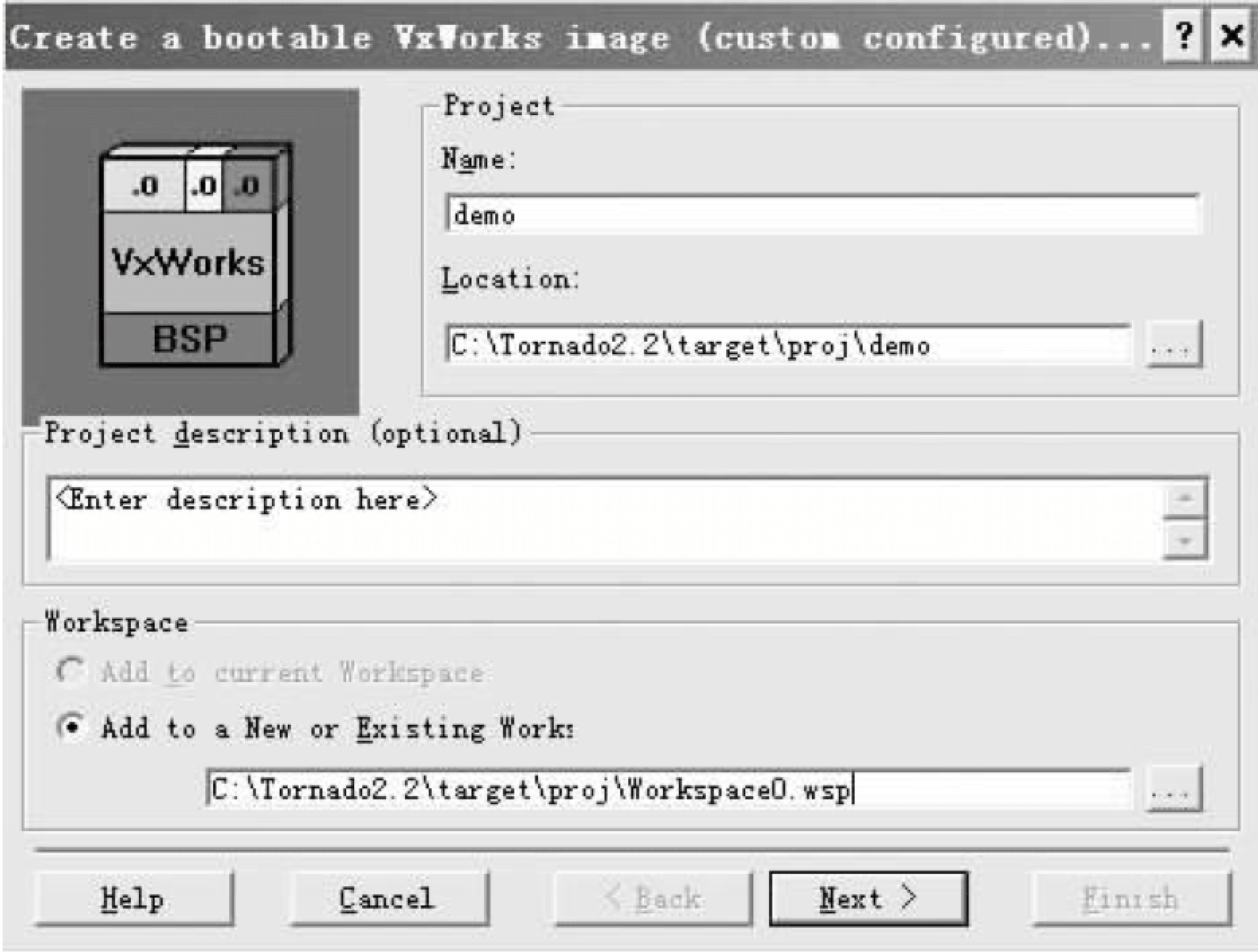



图 12.15 创建新的程序

(3) 编写源代码程序。利用 Tornado 的编译工具编写源代码程序，如图 12.16 所示。

(4) 下载并交互运行。在开发调试阶段后，选用手动下载模型映像进行仿真。当程序基本确定后，直接编译到 VxWorks 操作系统中，开机上电直接运行。

程序的运行主要包含 3 个步骤：

(1) 在主机和目标机之间建立通信连接。

(2) 将目标文件从主机转移到目标机上。

(3) 运行程序，即可获得系统仿真结果。

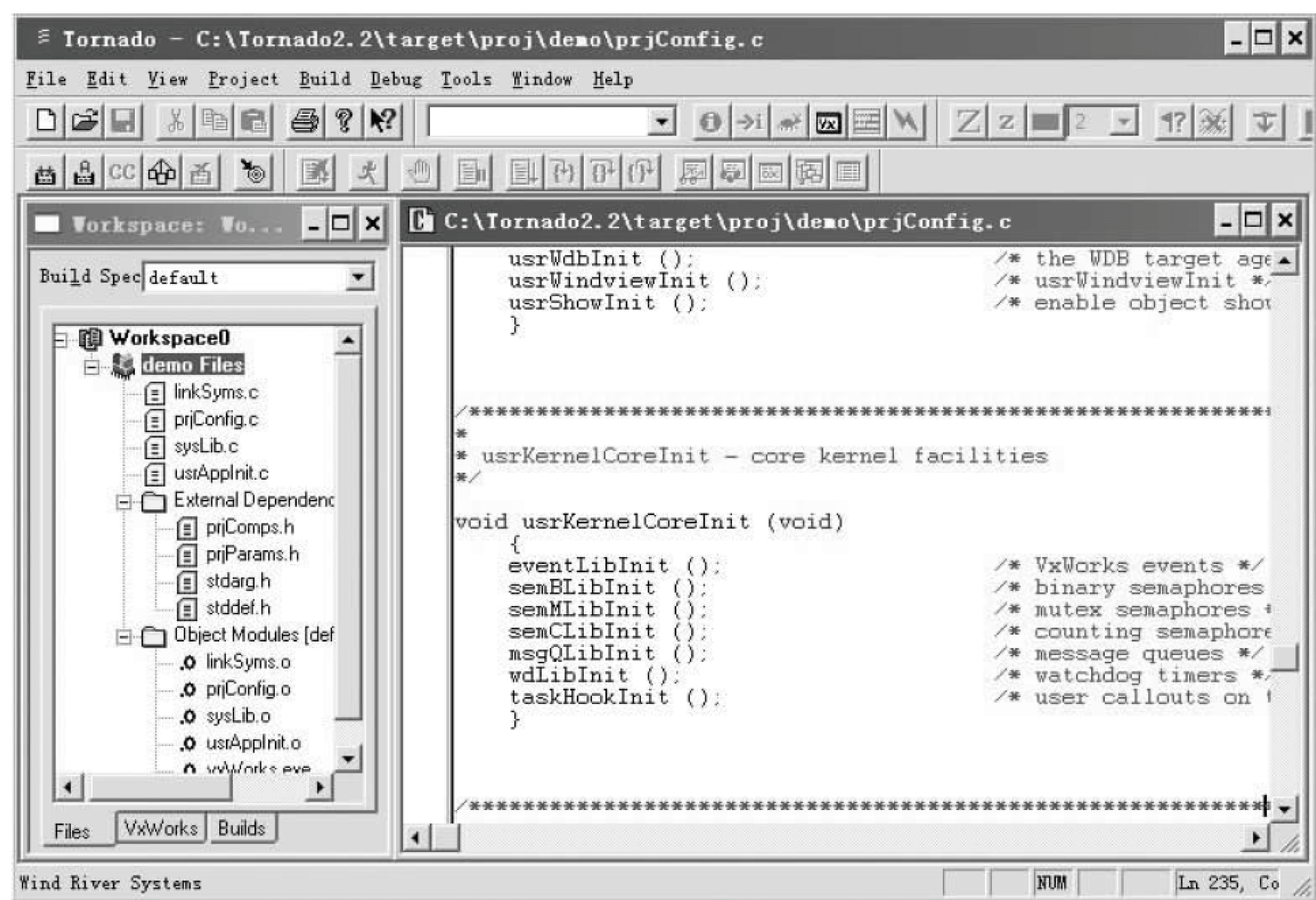



图 12.16　编写源代码

# 本 章 小 结

本章主要介绍了嵌入式操作系统的管理机制、嵌入式操作系统的集成开发环境及开发过程。

嵌入式操作系统是嵌入式系统的基础，是运行在嵌入式硬件平台上，对系统软硬件资源进行统一协调控制的操作系统软件。嵌入式操作系统具有高实时性、可裁剪性、高可靠性、接口统一、网络功能强大、体积小巧、固化代码、操作简单易学等特点，是各类嵌入式操作系统开发的统一无关性平台。

嵌入式操作系统采用多任务机制，根据完成截止时间的不同，实时任务分为硬实时任务和软实时任务。任务采用任务控制块来进行管理，分为就绪、执行和等待 3 个状态，任务要参与资源的竞争，只有在所需资源得到满足的情况下才能得到执行。通过任务调度，能使就绪态的任务获得 CPU 来执行，任务调度算法一般有基于优先级的抢占式调度算法和时间片轮转调度算法两种。基于优先级的抢占式调度是使具有最高优先级的任务先执行，时间片轮转调度算法主要是为相同优先级的任务调度服务，多数实时内核采用基于优先级调度的算法，但有些实时内核常常会实现两种算法混合的调度方案。由任务调度所带来的就是各个任务之间不可避免的相互协同关系，即任务间通信。一般嵌入式操作系统提供共享内存机制、信号量机制及消息队列与管道 3 种任务间通信方法。共享内存机制是通过访问共同的内存空间实现数据之间的相互传递；信号量机制是任务间少量信息的通信、提供同步和互斥的主要手段；而消息队列与管道是实现大量数据传输的重要手段。

嵌入式操作系统的内存管理机制采用动态内存管理机制和虚拟内存管理机制，操作系统提供了动态内存分配接口来完成动态内存管理。由于有些嵌入式系统中的 CPU 提供 MMU，MMU 具备内存地址映射和寻址功能，从而可以完成从虚拟地址到物理地址的转换，因此可以实现虚拟内存管理机制，这样，既可十分有效地提高系统可靠性和安全性，又可

满足程序的运行速度需求,节约物理内存空间。

I/O设备管理及文件系统也是嵌入式操作系统的重要组成部分。设备的管理分为缓冲技术、设备调度和设备分配3部分。中断处理是设备管理中至关重要的组成部分,中断是导致程序正常执行流程发生改变的事件。嵌入式操作系统中的文件管理和通用的操作系统中的文件管理有所不同,其主要提供文件存储、检索和更新等功能,一般不提供保护和加密等安全机制。在某些情况下,文件系统还可以针对特殊的目的来进行定制。

VxWorks是一种实时多任务的嵌入式操作系统,它由400多个相对独立的、短小精炼的目标模块组成,用户可根据需要选择适当模块来裁剪和配置系统,系统的链接器也可按应用的需要自动链接一些目标模块。VxWorks系统被广泛地应用与通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中,并已成为嵌入式系统开发中的主要开发环境之一。

随着嵌入式软件的发展,嵌入式操作系统的开发也越来越重要。Tornado集成开发环境是嵌入式软件开发中一种重要的交叉编译和调试环境,它以VxWorks为底层嵌入式操作系统,可以实现操作系统和嵌入式软件的共同开发,为嵌入式系统开发人员提供了一个不受目标机资源限制的超级开发和调试环境。

## 习　题

12.1　嵌入式操作系统的特点是什么?

12.2　嵌入式操作系统中任务的状态有哪些? 其状态之间是如何转换的?

12.3　什么叫任务切换? 任务切换的主要工作是什么?

12.4　嵌入式操作系统中如何实现任务间通信?

12.5　嵌入式操作系统中的内存管理机制如何设置?

12.6　嵌入式系统中的中断的分类有哪些? 中断处理的基本过程如何?

12.7　嵌入式操作系统中文件、设备和设备驱动之间的关系如何?

12.8　一般嵌入式系统的开发流程包括哪些步骤?

# 参 考 文 献


[1] Mark E. Russinovich, David A. Solomon. Microsoft Windows Internals. Microsoft Press, 2005.
[2] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Operating System Concepts. 6th Ed. 操作系统概念. 6 版. 郑扣根,译. 北京:高等教育出版社,2004.
[3] 陈向群,向勇,王雷,等. Windows 操作系统原理. 2 版. 北京:机械工业出版社,2004.
[4] Johnson M. Hart. Windows System Programming. 3rd Edition. Addison Wesley Professional, 2004.
[5] J. E. Smith, Ravi Nair. The architecture of virtual machines. IEEE Computer, 2005, 38(5): 32-38.
[6] A. S. Tanenbaum, A. S. Woodhull. Operating Systems: Design and Implementation. 2nd Ed. Prentice Hall, 1997.
[7] D. Milojicic, F. Douglis, R. Wheeler (eds). Mobility: Processes, Computers, and Agents. ACM Press, Addison-Wesley, 1999.
[8] J. L. Hennessy, D. Goldberg. Computer Architecture: A Quantitative Approach. 3rd Ed. Morgan Kaufmann, San Francisco, CA, 2003.
[9] H. Huang, P. Pillai, K. G. Shin. Design and implementation of power-aware virtual memory. In proc. USENIX 2003 Annual Technical Conference, San Antonio, Tx, June 2003: 57-70.
[10] Tauibamum A S. Modern Operating Systems. Englewood Cliffs: Prentice Hall,1992.
[11] Tauibamum A S,Woodhull A S. Operating Systems Design and implementation. Englewood Cliffs: Prentice Hall,1996.
[12] William Stalling. Operating Systems,Internals and Design principles. Englewood Cliffs: Prentice Hall,1998.
[13] 吴旭光,何军红. 嵌入式操作系统原理与应用. 北京:化学工业出版社,2007.